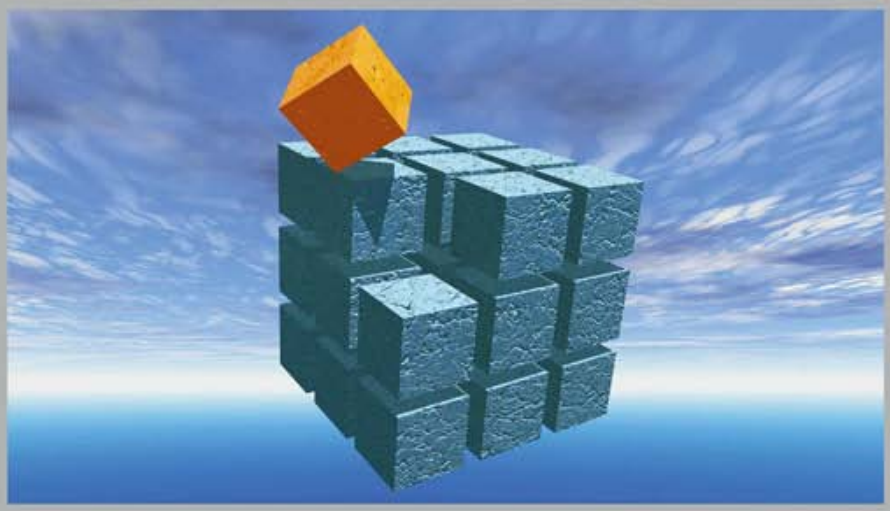
TEXTS IN COMPUTER SCIENCE

A Practical Introduction to Computer Architecture



Daniel Page

Springer

VISIT…

Texts in Computer Science

Editors
David Gries
Fred B. Schneider

For further volumes:
http://www.springer.com/series/3191

Daniel Page

A Practical Introduction to Computer Architecture

Springer

Dr. Daniel Page
University of Bristol
Dept. Computer Science
Room 3.10, Merchant Venturers Building
Woodland Road, Bristol
United Kingdom, BS8 1UB
page@compsci.bristol.ac.uk

Series Editors
David Gries
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501, USA

Fred B. Schneider
Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501, USA

ISBN 978-1-84882-255-9 e-ISBN 978-1-84882-256-6
DOI 10.1007/978-1-84882-256-6
Springer Dordrecht Heidelberg London New York

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2009922086

© Springer-Verlag London Limited 2009
Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licenses issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.
The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.
The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Kate ♡

Foreword

It is a great pleasure to write a preface to this book. In my view, the content is unique in that it blends traditional teaching approaches with the use of mathematics and a mainstream Hardware Design Language (HDL) as formalisms to describe key concepts. The book keeps the "machine" separate from the "application" by strictly following a bottom-up approach: it starts with transistors and logic gates and only introduces assembly language programs once their execution by a processor is clearly defined.

Using a HDL, Verilog in this case, rather than static circuit diagrams is a big deviation from traditional books on computer architecture. Static circuit diagrams cannot be explored in a hands-on way like the corresponding Verilog model can. In order to understand why I consider this shift so important, one must consider how computer architecture, a subject that has been studied for more than 50 years, has evolved.

In the pioneering days computers were constructed by hand. An entire computer could (just about) be described by drawing a circuit diagram. Initially, such diagrams consisted mostly of analogue components before later moving toward digital logic gates. The advent of digital electronics led to more complex cells, such as half-adders, flip-flops, and decoders being recognised as useful building blocks. However, miniaturisation of devices and hence computers has led to the design of single circuits containing millions or even billions of components. As a result, hand-lay-out is only used for specific modules, and circuit diagrams are less useful as a mechanism for describing functionality for real circuits.

Instead, two formalisms are used in industry: HDLs and mathematics. A HDL allows us to tell the component-layout and simulation tools how we would like to implement our circuit; mathematics tells us what the circuit ought to do. In order to verify whether the circuit does what we want it to, we can (partly mechanically) compare the mathematical description with the HDL description. This represents increased use of abstraction to cope with complexity, and an engineer can now be productive by simply understanding and using high-level circuit design (e.g., multiplier design or pipelined processors) and formalisms (e.g., HDLs and mathematics).

Circuit diagrams are still used in the design flow, but mostly to sketch the physical layout, in order to predict whether a circuit can be laid out sensibly.

Dealing with gaps in understanding between such a wide range of concepts and techniques is often off-putting for people new to the subject. The best way to approach the problem is by placing it within a practical context that enables students to experiment with ideas and discover themselves the advantages and disadvantages of a particular technique.

In this book, Dan does just that by giving an excellent overview of key concepts and an introduction to formalisms with which they can be explored. I hope this book will inspire many readers to follow a career in this fascinating subject.

Henk Muller.
Principal Technologist, XMOS.

Preface

In theory there is no difference between theory and practice. In practice there is.

– L.P. "Yogi" Berra

Overview

In my (limited) experience, and although there are a number of genuinely excellent textbooks on the subject, two main challenges exist in relation to delivery of University level taught modules in computer architecture:

1. Such modules are often regarded as unpopular and irrelevant by students who have not been exposed to the subject before, and who view a computer system from the applications level. This is compounded by the prevalence of technologies such as Java which place a further layer between the student and actual computer hardware. In short, and no matter how one tries to persuade them otherwise, students often see no point in learning about the internals of a computer system because they cannot see the benefit.
2. Conventional textbooks teach the subject in a different way than in other modules students are exposed to at the same time. For example, conventional wisdom says that one cannot "teach" programming, one has to "do" programming in order to learn. This is in stark contrast to textbooks on computer architecture where students are often forced to learn in a more theoretical way, learning by taking facts for granted rather than experimenting to arrive at their own conclusions. For example, because of the difficulty in working with large logic designs on paper, any practical work is often limited and hence detached from the more challenging content.

I would argue that this is a shame: computer architecture represents a broad spectrum of fundamental and exciting topics that underpin computer science in general. Aside from the technical challenges and sense of achievement that stem from

understanding exactly how high-level programs are actually executed on devices built from simple building blocks, historical developments in computer architecture neatly capture and explain many design decisions that have shaped a landscape we now take for granted. The representation of strings in C is a great example: the null-terminated ASCIIZ approach was not adopted for any real reason other than the PDP-7 computer included instructions ideal for processing strings in this form, and yet we still live with this decision years after the PDP-7 became obsolete. Seemingly frivolous anecdotes and examples like this are increasingly being consigned to history whereas from an Engineering perspective, one would like to learn and understand previous approaches so as to potentially improve in the future.

International experts regularly debate tools and techniques for delivering University-level modules in computer architecture; the Workshop on Computer Architecture Education (WCAE), currently held in conjunction with the International Symposium on Computer Architecture (ISCA), is the premier research conference in this area. This book represents an attempt at translating my personal philosophy, that theoretical concepts should be accessible for practical experimentation, into a form suitable for use in such modules. Put simply, I see computer architecture as a subject in which "getting things done" is paramount; the ability to understand trade-offs before selecting between and implementing well considered design options is often as important as the study of those options at a more theoretical level. This focus is underlined by the book sub-title: a "practical" approach is the aim throughout. To enable this, a key feature of this book is inclusion and use of a hardware description language (i.e., Verilog), and a concrete processor (i.e., MIPS32) as practical vehicles for modelling and experimenting with digital logic and processor design.

Target Audience

The content is organised into three parts which contain a total of thirteen core chapters. Although some slight disagreement about inclusion of specific topics is inevitable, the chapters represent a compromise between my informal opinion, interests and experience, and more formal curriculum guidelines such as that developed by the UK Quality Assurance Agency (QAA):

`http://www.qaa.ac.uk/`

Of course, international equivalents exist; examples include those developed jointly by the IEEE and ACM, leading professional bodies within this domain:

`http://www.computer.org/curriculum/`

The general aim of this book is to cover topics *every* computer science student should have at least a basic grasp of, and equip said students with enough knowledge to read and understand more advanced textbooks. In this respect, the core target audience is first-year Undergraduate students with a rudimentary knowledge of

programming in C. More generally, the book content is pitched at a level which satisfies most of the demands that have resulted from our degree programmes at the University of Bristol. In particular, the more advanced material has proved useful as a bridge toward, or in support of, more specialised textbooks that cover later-year Undergraduate and Postgraduate modules.

Organisation

The book chapters are described briefly below. Very roughly the three parts of the book can be viewed as somewhat self-contained, representing three layers or levels of abstraction: the **digital logic layer**, the **instruction set and micro-architecture layer**, and the **hardware/software interface**. Part 1 deals with basic tools and techniques which underpin the rest of the book:

Chapter 1 Introduces the theoretical background including logic, sets, and number representation.

Chapter 2 Uses the theory developed in the previous chapter to describe the basics of digital logic including logic gates and their construction using transistors, combinatorial and clocked circuits and their optimisation.

Chapter 3 As a means of realising the digital logic designs presented in the previous chapter, Verilog is presented in an introductory manner; this content is written with a reader who is a proficient C programmer in mind.

Part 2 deals with the broad topic of processor design and implementation. The content takes a step-by-step approach, starting with a functional description of a computer processor and gradually expanding on the details, issues and techniques that have resulted in modern, high-performance processor designs:

Chapter 4 The first step in introducing the design of processors (as an extension to the study of general circuits) is to track historical developments and use them as a means to explain central concepts such as the fetch-decode-execute cycle.

Chapter 5 Once the core concepts are introduced, a concrete realisation in the form of MIPS32 is discussed; this discussion includes details such as addressing modes and instruction encoding for example.

Chapter 6 This chapter outlines some basic methods for evaluating processors in terms of various metrics which can be used to defined quality, focusing on performance in particular.

Chapter 7 As a vital component in any processor, the design of efficient circuits for arithmetic (e.g., addition and multiplication) is introduced and demonstrated using Verilog.

Chapter 8 In the same way as arithmetic, the design of efficient components in the memory hierarchy is introduced and demonstrated using Verilog.

Chapter 9 Finally, a number of more advanced topics in processor design are investigated including approaches such as superscalar and vector processors.

Finally, Part 3 attempts to bridge the gap between hardware and software by examining the programming tools and operating system concepts that support the development and execution of programs:

Chapter 10 As a first step, this chapter presents a detailed description of a development tool-chain, starting with linkers and assemblers. This material is supported by and links to Appendix A, which provides a stand-alone tutorial on using SPIM, a MIPS32 simulator. As such, it provides a concrete means of writing programs for the processor design introduced earlier in the book.

Chapter 11 Following the previous chapter, compilers are now introduced by focusing on their aspects which are most closely tied to the processor they target. For example, register allocation, instruction select and scheduling are all covered.

Chapter 12 With a means of writing programs established, the operating system layer is introduced in a practical way by using SPIM as a platform for real implementations of concepts such as scheduling and interrupt handling.

Chapter 13 Finally, a somewhat novel chapter examines the concept of efficient programming. This is written from the point of view of a programmer who wants (or needs) to capitalise on the behaviour and characteristics of the concepts presented in previous chapters to improve their programs.

Each chapter concludes with a set of example questions which are largely at a level one might encounter in a degree-level examination. These questions are numbered consecutively (rather than relative to the chapter number) and a set of example solutions can be found in Appendix B.

Although this content might clearly contain accidental omissions, some topics have been avoided by design. Optimistically, one might view them as ideal for inclusion in future versions of the book; more realistically they represent topics that are important but not vital at the level the book is aimed at:

- Perhaps the largest omission, at least the one which seems likely to prompt the loudest outcry, is that of **floating point arithmetic**. Although the book covers floating point representation briefly, the circuits for arithmetic and instruction sets that use them are integer only. The rationale for this decision was purely space: floating point represents a fairly self-contained topic which could be left out without too negative an impact on core topics covered by the rest of the content.
- A similar situation exists with the topic of **design verification**: although testing the digital logic designs one can produce with Verilog is important, one could easily dedicate an entire book to the art of good design verification.
- The main emphasis of the book is processors and as such, they are examined somewhat in isolation. This contrasts with reality where such processors typically form part of a larger system including peripherals, communication networks and so on. As such, the broad topic of **system-level** design, which is central to other books, is not covered here.
- At the time of writing, the "hot topic" in computer architecture is the advent of **multi-core** devices, i.e., many processors on a single chip. Although this has been

a research area for many years in an attempt to cope with design complexity and effective use of increased transistor counts, commodity multi-core devices have breathed new life into the subject. Again, because the emphasis is more intra-processor than inter-processor, we omit the topic here although among all the options for future material this is perhaps the most compelling.

- The book covers only traditional logic styles. However, for specific application areas it has become attractive to investigate alternatives; the idea is basically that one should be aware of available alternatives and use the right one in the right place. Specifically, **secure logic styles** (which help to prevent certain types of passive attack) and **asynchronous logic** (which avoids the need for global clock signals) represent interesting avenues for future content.

Contact

As people who know me will (too) willingly attest, I am far from perfect. As a result, this book is sure to include problems in the shape of minor errors and mistakes. If you find such a problem, or have a more general comment, I would be glad to hear about it; you can contact me via

`http://www.cs.bris.ac.uk/home/page/`

Acknowledgements

This book was typeset with LaTeX, originally developed by Leslie Lamport and based on TeX by Donald Knuth; the `listings` package by Carsten Heinz, the `algorithm2e` package by Christophe Fiorio and `karnaugh` by Andreas Wieland were all used in addition to the basic system. The various figures and source code listing were generated with the help of `ModelSim`, a Verilog simulator by Mentor Graphics; `GTKWave`, a VCD waveform viewer by Anthony Bybell; `xfig`, a general vector drawing package originally written by Supoj Sutanthavibul and maintained by many others; `xcircuit`, a circuit vector drawing package by Tim Edwards; `asymptote`, a scripted vector drawing package by Andy Hammerlindl, John Bowman, and Tom Prince; SPIM, a MIPS32 simulator written by James Larus; Dinero, a cache simulator written by Jan Edler and Mark Hill; and finally `gcc` the GNU C compiler, originally by Richard Stallman and now maintained by many others.

Throughout the book, images from other sources are reproduced under specific licenses; each image of this type carefully notes the source and the license in question. Specific details of the GNU Free Documentation License and Creative Commons Licenses can be found via

`http://www.gnu.org/licenses/`

and

http://creativecommons.org/licenses/

Of course, as with any project, numerous people contributed in other ways. I would like to thank the (extended) Page, Symonds, Gould and Hunkin families and my friends Stan and Paul, Gavin and Heather and Fry's Hockey Club for their support, and providing an escape from work; Maisie, you still owe me a "Busy Bee" for helping with your homework. The book could not have been completed without the help and guidance of Wayne Wheeler, Catherine Brett and Simon Rees at Springer-Verlag. It probably would not have been started at all without the encouragement and tutelage of Nigel Smart, Henk Muller, David May and James Irwin within the Computer Science Department at the University of Bristol. Staff and students in the Cryptography and Information Security Group have provided a constant sounding board for ideas; I would like to thank Andrew Moss, Rob Granger, Philipp Grabher and Johann Großschädl in particular. Like all good Engineers, students at the University of Bristol have never been shy to say when I am talking nonsense or present something in a particularly boring way; my thanks to them and many anonymous reviewers for improving the editorial quality throughout.

Most of all I thank Kate for making it all worthwhile.

Contents

Part I Tools and Techniques

1 Mathematical Preliminaries 3
1.1 Propositions and Predicates 3
1.1.1 Connectives 5
1.1.2 Quantifiers 7
1.1.3 Manipulation 8
1.2 Sets and Functions 10
1.2.1 Construction 11
1.2.2 Operations 12
1.2.3 Numeric Sets 14
1.2.4 Functions 14
1.2.5 Relations 17
1.3 Boolean Algebra 18
1.3.1 Boolean Functions 21
1.3.2 Normal Forms 22
1.4 Number Systems 23
1.4.1 Converting Between Bases 25
1.4.2 Bits, Bytes and Words 27
1.4.3 Representing Numbers and Characters 29
1.5 Toward a Digital Logic 40
1.6 Further Reading 41
1.7 Example Questions 42

2 Basics of Digital Logic 43
2.1 Switches and Transistors 43
2.1.1 Basic Physics 43
2.1.2 Building and Packaging Transistors 44
2.2 Combinatorial Logic 48
2.2.1 Basic Logic Gates 48
2.2.2 3-state Logic 50

2.2.3 Designing Circuits 51
2.2.4 Simplifying Circuits 53
2.2.5 Physical Circuit Properties 61
2.2.6 Basic Building Blocks 63
2.3 Clocked and Stateful Logic 74
2.3.1 Clocks 75
2.3.2 Latches 77
2.3.3 Flip-Flops 80
2.3.4 State Machines 81
2.4 Implementation and Fabrication Technologies 87
2.4.1 Silicon Fabrication 87
2.4.2 Programmable Logic Arrays 90
2.4.3 Field Programmable Gate Arrays 92
2.5 Further Reading 93
2.6 Example Questions 93

3 Hardware Design Using Verilog 97
3.1 Introduction 97
3.1.1 The Problem of Design Complexity 97
3.1.2 Design Automation as a Solution 98
3.2 Structural Design 100
3.2.1 Modules 100
3.2.2 Wires 101
3.2.3 Values and Constants 103
3.2.4 Comments 104
3.2.5 Basic Instantiation 105
3.2.6 Nested Instantiation 108
3.2.7 User-Defined Primitives 108
3.3 Higher-level Constructs 109
3.3.1 Continuous Assignments 110
3.3.2 Selection and Concatenation 112
3.3.3 Reduction 114
3.3.4 Timing and Delays 114
3.4 State and Clocked Design 115
3.4.1 Registers 115
3.4.2 Processes and Triggers 117
3.4.3 Procedural Assignments 118
3.4.4 Timing and Delays 120
3.4.5 Further Behavioural Statements 121
3.4.6 Tasks and Functions 126
3.5 Effective Development 127
3.5.1 System Tasks 128
3.5.2 Using the Pre-processor 129
3.5.3 Parameters 131
3.5.4 Named Port Lists 132

3.5.5 Generate Statements . . . 133
3.5.6 Simulation and Stimuli . . . 134
3.6 Further Reading . . . 136
3.7 Example Questions . . . 137

Part II Processor Design

4 A Historical and Functional Perspective . . . 143
4.1 Introduction . . . 143
4.2 Special-Purpose Computers . . . 144
4.3 General-Purpose Computers . . . 146
4.4 Stored Program Computers . . . 150
4.5 Toward Modern Computers . . . 158
4.5.1 The von Neumann Bottleneck . . . 160
4.5.2 Data-Dependent Control-Flow . . . 160
4.5.3 Self-Modifying Programs . . . 161
4.6 Further Reading . . . 166
4.7 Example Questions . . . 166

5 Basic Processor Design . . . 169
5.1 A Concrete Stored Program Architecture . . . 169
5.1.1 Major Data-path Components . . . 171
5.1.2 Describing Instruction Behaviour . . . 174
5.1.3 The Fetch-Decode-Execute Cycle . . . 176
5.1.4 Controlling the Data-path . . . 176
5.2 Buses . . . 177
5.2.1 Synchronous Buses . . . 177
5.2.2 Asynchronous Buses . . . 178
5.3 Addressing Modes . . . 179
5.3.1 Immediate Addressing . . . 180
5.3.2 Register Addressing . . . 180
5.3.3 Memory Addressing . . . 180
5.4 Instruction Encoding . . . 183
5.4.1 Instruction Selection . . . 183
5.4.2 Instruction Formats . . . 186
5.4.3 Basic Encoding and Decoding . . . 186
5.4.4 More Complicated Encoding Issues . . . 189
5.5 Control-Flow . . . 192
5.5.1 Predicated Execution . . . 194
5.5.2 Function Calls . . . 197
5.6 Some Design Philosophy . . . 198
5.6.1 Moore's Law . . . 198
5.6.2 RISC versus CISC . . . 199
5.7 Putting It All Together . . . 200
5.8 Further Reading . . . 208
5.9 Example Questions . . . 209

6 Measuring Performance 213
6.1 Measuring Performance 213
6.1.1 Estimating Execution Time 214
6.1.2 Measuring Execution Time 216
6.1.3 Benchmark Programs 218
6.1.4 Measuring Improvement 219
6.2 Further Reading 220
6.3 Example Questions 220

7 Arithmetic and Logic 223
7.1 Introduction 223
7.2 Comparisons 224
7.2.1 Unsigned Comparisons 225
7.2.2 Signed Comparisons 228
7.3 Addition and Subtraction 228
7.3.1 Addition 228
7.3.2 Subtraction 233
7.4 Shift and Rotate 236
7.4.1 Bit-Serial Shifter 238
7.4.2 Logarithmic Shifter 241
7.5 Multiplication 243
7.5.1 Bit-Serial Multiplier 245
7.5.2 Tree Multiplier 249
7.5.3 Digit-Serial Multiplier 250
7.5.4 Early Termination 251
7.5.5 Wallace and Dadda Trees 253
7.5.6 Booth Recoding 257
7.6 Putting It All Together 259
7.6.1 Comparison ALU 259
7.6.2 Arithmetic ALU 262
7.7 Further Reading 265
7.8 Example Questions 266

8 Memory and Storage 269
8.1 Introduction 269
8.1.1 Historical Memory and Storage 271
8.1.2 A Modern Memory Hierarchy 277
8.1.3 Basic Organisation and Implementation 279
8.1.4 Memory Banking 288
8.1.5 Access Locality 290
8.2 Memory and Storage Specifics 291
8.2.1 Static RAM (SRAM) and Dynamic RAM (DRAM) 291
8.2.2 Non-volatile RAM and ROM 293
8.2.3 Magnetic Disks 294

8.2.4 Optical Disks . . . 296
8.2.5 Error Correction . . . 297
8.3 Basic Cache Memories . . . 300
8.3.1 Fetch Policy . . . 303
8.3.2 Write Policy . . . 304
8.3.3 Direct-Mapped Caches . . . 305
8.3.4 Fully-Associative Caches . . . 310
8.3.5 Set-Associative Caches . . . 312
8.3.6 Cache Organisation . . . 315
8.4 Advanced Cache Memories . . . 316
8.4.1 Victim Caches . . . 316
8.4.2 Gated and Drowsy Caches . . . 319
8.5 Putting It All Together . . . 321
8.5.1 Register File . . . 322
8.5.2 Main Memory . . . 323
8.5.3 Cache Memory . . . 324
8.6 Further Reading . . . 329
8.7 Example Questions . . . 329

9 Advanced Processor Design . . . 331
9.1 Introduction . . . 331
9.1.1 A Taxonomy of Parallelism . . . 332
9.1.2 Instruction-Level Parallelism (ILP) . . . 335
9.2 Pipelined Processors . . . 339
9.2.1 Pipelined Circuits . . . 343
9.2.2 Pipelined Processors . . . 347
9.2.3 Pipeline Hazards . . . 350
9.2.4 Stalls and Hazard Resolution . . . 352
9.3 Superscalar Processors . . . 360
9.3.1 Basic Concept . . . 360
9.3.2 Step 1: Scoreboard-based Design . . . 361
9.3.3 Step 2: Reservation Station-based Design . . . 370
9.3.4 Further Improvements . . . 378
9.4 Vector Processors . . . 380
9.4.1 Basic Concept . . . 380
9.4.2 A Dedicated Vector Processor . . . 382
9.4.3 SIMD Within A Register (SWAR) . . . 384
9.4.4 Issues of Vectorisation . . . 386
9.5 VLIW Processors . . . 389
9.5.1 Basic Concept . . . 389
9.6 Further Reading . . . 390
9.7 Example Questions . . . 390

Part III The Hardware/Software Interface

10 Linkers and Assemblers 397
10.1 Introduction 397
10.2 The Memory Model 400
10.2.1 Stack Section 401
10.2.2 Static Data Section 402
10.2.3 Dynamic Data Section 403
10.3 Executable Versus Object Files 407
10.4 Linkers 410
10.4.1 Static and Dynamic Linkage 413
10.4.2 Boot-strap Functions 415
10.4.3 Symbol Relocation 416
10.4.4 Symbol Resolution 417
10.5 Assemblers 417
10.5.1 Basic Assembly Language Statements 418
10.5.2 Using Machine Instructions 420
10.5.3 Using Assembler Aliases 429
10.5.4 Using Assembler Directives 431
10.5.5 Peephole Optimisation 434
10.5.6 Some Short Example Programs 435
10.5.7 The Forward Referencing Problem 439
10.5.8 An Example Assembler 441
10.6 Further Reading 449
10.7 Example Questions 449

11 Compilers 451
11.1 Introduction 451
11.2 Compiler Bootstrapping and Re-Hosting 453
11.3 Intermediate Representation 454
11.4 Register Allocation 457
11.4.1 An Example Allocation 461
11.4.2 Uses for Pre-colouring 463
11.4.3 Avoiding Error Cases via Spilling 464
11.5 Instruction Selection and Scheduling 467
11.5.1 Instruction Selection 468
11.5.2 Instruction Scheduling 470
11.5.3 Scheduling Basic Blocks 470
11.5.4 Scheduling Instructions 471
11.6 “Template” Code Generation for High-Level Statements 473
11.6.1 Conditional Statements 475
11.6.2 Loop Statements 477
11.6.3 Multi-way Branch Statements 478
11.7 “Template” Code Generation for High-Level Function Calls 479
11.7.1 Basic Stack Frames 481

11.7.2 Advanced Stack Frames . . . 485
11.8 Further Reading . . . 491
11.9 Example Questions . . . 492

12 Operating Systems . . . 495
12.1 Introduction . . . 495
12.2 The Hardware/Software Interface . . . 497
12.2.1 MIPS32 Co-processor Registers . . . 497
12.2.2 MIPS32 Processor Modes . . . 499
12.2.3 MIPS32 Assembly Language . . . 500
12.3 Boot-Strapping . . . 501
12.4 Event Management . . . 502
12.4.1 Handling Interrupts . . . 503
12.4.2 Handling Exceptions . . . 505
12.4.3 Handling Traps . . . 506
12.4.4 An Example Exception Handler . . . 510
12.5 Memory Management . . . 511
12.5.1 Basic Concept . . . 512
12.5.2 Pages and Frames . . . 515
12.5.3 Address Translation and Memory Access . . . 518
12.5.4 Page Eviction and Replacement . . . 520
12.5.5 Translation Look-aside Buffer (TLB) . . . 521
12.6 Process Management . . . 522
12.6.1 Storing and Switching Process Context . . . 523
12.6.2 Process Scheduling . . . 524
12.6.3 An Example Scheduler . . . 529
12.7 Further Reading . . . 533
12.8 Example Questions . . . 534

13 Efficient Programming . . . 535
13.1 Introduction . . . 535
13.2 "Space" Conscious Programming . . . 536
13.2.1 Reducing Register Pressure . . . 536
13.2.2 Reducing Memory Allocation . . . 537
13.3 "Time" Conscious Programming . . . 540
13.3.1 Effective Short-circuiting . . . 540
13.3.2 Branch Elimination . . . 541
13.3.3 Loop Fusion and Fission . . . 542
13.3.4 Loop Unrolling . . . 545
13.3.5 Loop Hoisting . . . 546
13.3.6 Loop Interchange . . . 548
13.3.7 Loop Blocking . . . 549
13.3.8 Function Inlining . . . 551
13.3.9 Software Pipelining . . . 554
13.4 Example Questions . . . 556

Part IV Appendices

SPIM: A MIPS32 Simulator 561
A.1 Introduction 561
A.2 Configuring SPIM 562
A.3 Controlling SPIM 563
A.4 Example Program Execution 565
A.5 Using System Calls 570

Example Solutions 573

References 629

Index 633

Part I
Tools and Techniques

Chapter 1
Mathematical Preliminaries

In mathematics you don't understand things. You just get used to them.

– J. von Neumann

Abstract The goal of this chapter is to give a fairly comprehensive overview of the theory that underpins the rest of the book. On first reading, it may seem a little dry and is often excluded in other similar books. However, without a solid understanding of logic and representation of numbers it seems clear that constructing digital circuits to put this theory into practise would be much harder. The theory here will present an introduction to propositional logic, sets and functions, number systems and Boolean algebra. These four main areas combine to produce a basis for formal methods to describe, manipulate and implement digital systems such as computer processors. Those with a background in mathematics or computer science might skip this material and use it simply for reference; those approaching the subject from another background would be advised to read the material in more detail.

1.1 Propositions and Predicates

Definition 1. A **proposition** is a statement whose meaning, termed the **truth value**, is either true or false. Less formally, we say the statement is true if it has a truth value of true and false if it has a truth value of false.

A **predicate** is a proposition which contains one or more **variables**; only when concrete values are **assigned** to each of the variables can the predicate be called a proposition.

Since we use them so naturally, it almost seems too formal to define what a proposition is. However, by doing so we can start to use them as a building block to describe what logic is and how it works. The statement

"the temperature is $90^{\circ}C$"

is a proposition since it is definitely either true or false. When we take a proposition and decide whether it is true or false, we say we have **evaluated** it. However, there are clearly a lot of statements that are not propositions because they do not state any proposal. For example,

"turn off the heat"

is a command or request of some kind, it does not evaluate to a truth value. Propositions must also be well defined in the sense that they are definitely either true or false, i.e., there are no "gray areas" in between. The statement

"$90^\circ C$ is too hot"

is not a proposition since it could be true or false depending on the context, or your point of view: $90^\circ C$ probably is too hot for body temperature but probably not for a cup of coffee. Finally, some statements look like propositions but cannot be evaluated because they are paradoxical. The most famous example of this situation is the liar paradox, usually attributed to the Greek philosopher Eubulides, who stated it as

"a man says that he is lying, is what he says true or false ?"

although a clearer version is the more commonly referenced

"this statement is false" .

If the man is telling the truth, everything he says must be true which means he is lying and hence everything he says is false. Conversely, if the man is lying everything he says is false, so he cannot be lying since he said he was ! In terms of the statement, we cannot be sure of the truth value so this is not normally classed as a proposition.

As stated above, a predicate is just a proposition that contains variables. By **assigning** the variable a value we can turn the predicate into a proposition and evaluate the corresponding truth value. For example, consider the predicate

"$x^\circ C$ equals $90^\circ C$"

where x is a variable. By assigning x a value we get a proposition; setting $x = 10$, for example, gives

"$10^\circ C$ equals $90^\circ C$"

which clearly evaluates to false. Setting $x = 90^\circ C$ gives

"$90^\circ C$ equals $90^\circ C$"

which evaluates to true. In some sense, a predicate is an abstract or general proposition which is not well defined until we assign values to all the variables.

1.1.1 Connectives

Definition 2. A **connective** is a statement which binds single propositions into a compound proposition. For brevity, we use symbols to denote common connectives:

- "not x" is denoted $\neg x$.
- "x and y" is denoted $x \wedge y$.
- "x or y" is denoted $x \vee y$, this is usually called an inclusive-or.
- "x or y but not x and y" is denoted $x \oplus y$, this is usually called an exclusive-or.
- "x implies y" is denoted $x \rightarrow y$, which is sometimes written as "if x then y".
- "x is equivalent to y" is denoted $x \leftrightarrow y$, which is sometimes written as "x if and only if y" or further shortened to "x iff. y".

Note that we group statements using parentheses when there could be some confusion about the order they are applied; hence $(x \wedge y)$ is the same as $x \wedge y$.

A proposition or predicate involving connectives is built from **terms**; the connective joins together these terms into an **expression**. For example, the expression

"the temperature is less than $90°C$ $\wedge$ the temperature is greater than $10°C$"

contains two terms that propose

"the temperature is less than $90°C$"

and

"the temperature is greater than $10°C$" .

These terms are joined together using the $\wedge$ connective so that the whole expression evaluates to true if both of the terms are true, otherwise it evaluates to false. In a similar way we might write a compound predicate

"the temperature is less than $x°C$ $\wedge$ the temperature is greater than $y°C$"

which can only be evaluated when we assign values to the variables x and y.

Definition 3. The meaning of connectives is usually describe in a tabular form which enumerates the possible values each term can take and what the resulting truth value is; we call this a **truth table**.

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \rightarrow y$	$x \leftrightarrow y$
false	false	true	false	false	false	true	true
false	true	true	false	true	true	true	false
true	false	false	false	true	true	false	false
true	true	false	true	true	false	true	true

The $\neg$ connective negates the truth value of an expression so considering

$$\neg(x > 10)$$

we find that the expression $\neg(x > 10)$ is true if the term $x > 10$ is false and the expression is false if $x > 10$ is true. If we assign $x = 9$, $x > 10$ is false and hence the expression $\neg(x > 10)$ is true. If we assign $x = 91$, $x > 10$ is true and hence the expression $\neg(x > 10)$ is false.

The meaning of the $\wedge$ connective is also as one would expect; the expression

$$(x > 10) \wedge (x < 90)$$

is true if *both* the expressions $x > 10$ and $x < 90$ are true, otherwise it is false. So if $x = 20$, the expression is true. But if $x = 9$ or $x = 91$, then it is false: even though one or other of the terms is true, they are not both true.

The inclusive-or and exclusive-or connectives are fairly similar. The expression

$$(x > 10) \vee (x < 90)$$

is true if *either* $x > 10$ *or* $x < 90$ is true or both of them are true. Here we find that all the assignments $x = 20$, $x = 9$ and $x = 91$ mean the expression is true; in fact it is hard to find an x for which it evaluates to false ! Conversely, the expression

$$(x > 10) \oplus (x < 90)$$

is only true if *only one* of either $x > 10$ or $x < 90$ is true; if they are both true then the expression is false. We now find that setting $x = 20$ means the expression is false while both $x = 9$ and $x = 91$ mean it is true.

Inference is a bit more tricky. If we write $x \rightarrow y$, we usually call x the **hypothesis** and y the **conclusion**. To justify the truth table for inference in words, consider the example

$$(x \text{ is prime }) \wedge (x \neq 2) \rightarrow (x \equiv 1 \pmod 2)$$

i.e., if x is a prime other than 2, it follows that it is odd. Therefore, if x is prime then the expression is true if $x \equiv 1 \pmod 2$ and false otherwise since the implication is invalid. If x is not prime, the expression does not really say anything about the expected outcome: we only know what to expect if x was prime. Since it could still be that $x \equiv 1 \pmod 2$ even when x is not prime, and we do not know anything better from the expression, we assume it is true when this case occurs.

Equivalence is fairly simple. The expression $x \leftrightarrow y$ is only true if x and y evaluate to the same value. This matches the idea of equality of numbers. As an example, consider

$$(x \text{ is odd }) \leftrightarrow (x \equiv 1 \pmod 2).$$

This expression is true since if the left side is true, the right side must also be true and vice versa. If we change it to

$$(x \text{ is odd }) \leftrightarrow (x \text{ is prime }),$$

then the expression is false. To see this, note that only some odd numbers are prime: just because a number is odd does not mean it is always prime although if it is prime

it must be odd (apart from the corner case of $x = 2$). So the equivalence works in one direction but not the other and hence the expression is false.

Definition 4. We call two expressions logically **equivalent** if they are composed of the same variables and have the same truth value for every possible assignment to those variables.

An expression which is equivalent to true, no matter what values are assigned to any variables, is called a **tautology**; an expression which is equivalent to false is called a **contradiction**.

Some subtleties emerge when trying to prove two expressions are logically equivalent. However, we will skirt around these. For our purposes it suffices to simply enumerate all possible values each variable can take, and check the two expressions produce identical truth values in all cases. In practise this can be hard since with n variables there will be 2^n possible assignments, an amount which grows quickly as n grows ! More formally, two expressions x and y are only equivalent if $x \leftrightarrow y$ can be proved a tautology.

1.1.2 Quantifiers

Definition 5. A **free variable** in a given expression is one which has not yet been assigned a value. Roughly speaking, a **quantifier** is a statement which allows a free variable to take one of many values:

- the **universal quantifier** "for all x, y is true" is denoted $\forall\, x\, [y]$.
- the **existential quantifier** "there exists an x such that y is true" is denoted $\exists\, x\, [y]$.

We say that applying a quantifier to a variable quantifies it; after it has been quantified we say it has been **bound**.

Put more simply, when we encounter an expression such as

$$\exists\, x\, [y]$$

we are essentially assigning x all possible values; to make the expression true *just one* of these values needs to make the expression y true. Likewise, when we encounter

$$\forall\, x\, [y]$$

we are again assigning x all possible values. This time however, to make the expression true, *all of them* need to make the expression y true. Consider the following example:

$$\text{"there exists an } x \text{ such that } x \equiv 0 \pmod 2\text{"}$$

which we can re-write symbolically as

$$\exists\, x\ [x \equiv 0 \pmod 2].$$

In this case, x is bound by the $\exists$ quantifier; we are asserting that for some value of x it is true that $x \equiv 0 \pmod 2$. To make the expression true just one of these values needs to make the term $x \equiv 0 \pmod 2$ true. The assignment $x = 2$ satisfies this so the expression is true. As another example, consider the expression

$$\text{“for all } x,\ x \equiv 0 \pmod 2\text{”}$$

which we re-write

$$\forall\, x\ [x \equiv 0 \pmod 2].$$

Here we are making a more general assertion about x by saying that for all x, it is true that $x \equiv 0 \pmod 2$. To decide if this particular expression is false, we need simply to find an x such that $x \not\equiv 0 \pmod 2$. This is easy since any odd value of x is good enough. Therefore the expression is false.

Definition 6. Informally, a **predicate function** is just a shorthand way of writing predicates; we give the function a name and a list of free variables. So for example the function

$$f(x,y) : x = y$$

is called f and has two variables named x and y. If we use the function as $f(10,20)$, performing the binding $x = 10$ and $y = 20$, it has the same meaning as $10 = 20$.

1.1.3 Manipulation

Definition 7. Manipulation of expressions is governed by a number of axiomatic laws:

equivalence	$x \leftrightarrow y$	$\equiv (x \rightarrow y) \wedge (y \rightarrow x)$
implication	$x \rightarrow y$	$\equiv \neg x \vee y$
involution	$\neg\neg x$	$\equiv x$
idempotency	$x \wedge x$	$\equiv x$
commutativity	$x \wedge y$	$\equiv y \wedge x$
association	$(x \wedge y) \wedge z$	$\equiv x \wedge (y \wedge z)$
distribution	$x \wedge (y \vee z)$	$\equiv (x \wedge y) \vee (x \wedge z)$
de Morgan's	$\neg(x \wedge y)$	$\equiv \neg x \vee \neg y$
identity	$x \wedge \text{true}$	$\equiv x$
null	$x \wedge \text{false}$	$\equiv \text{false}$
inverse	$x \wedge \neg x$	$\equiv \text{false}$
absorption	$x \wedge (x \vee y)$	$\equiv x$
idempotency	$x \vee x$	$\equiv x$
commutativity	$x \vee y$	$\equiv y \vee x$
association	$(x \vee y) \vee z$	$\equiv x \vee (y \vee z)$
distribution	$x \vee (y \wedge z)$	$\equiv (x \vee y) \wedge (x \vee z)$
de Morgan's	$\neg(x \vee y)$	$\equiv \neg x \wedge \neg y$
identity	$x \vee \text{false}$	$\equiv x$
null	$x \vee \text{true}$	$\equiv \text{true}$
inverse	$x \vee \neg x$	$\equiv \text{true}$
absorption	$x \vee (x \wedge y)$	$\equiv x$

A common reason to manipulate a logic expression is to simplify it in some way so that it contains less terms or less connectives. A simplified expression might be more opaque, in the sense that it is not as clear what it means, but looking at things computationally it will generally be "cheaper" to evaluate. As a concrete example of simplification in action, consider the exclusive-or connective $x \oplus y$ which we can write as the more complicated expression

$$(y \wedge \neg x) \vee (x \wedge \neg y)$$

or even as

$$(x \vee y) \wedge \neg(x \wedge y).$$

The first alternative above uses five connectives while the second uses only four; we say that the second is simpler as a result. One can prove that these are equivalent by constructing truth tables for them. The question is, how did we get from one to the other by manipulating the expressions ?

To answer this, we simply start with one expression and apply our axiomatic laws to move toward the other. So starting with the first alternative, we try to apply the axioms until we get the second: think of the axioms like rules that allow one to re-write the expression in a different way. To start with, we can manipulate each term which looks like $p \wedge \neg q$ as follows

$$\begin{aligned}
(p \wedge \neg q) &= (p \wedge \neg q) \vee \text{false} && \text{(identity)} \\
&= (p \wedge \neg q) \vee (p \wedge \neg p) && \text{(null + inverse)} \\
&= p \wedge (\neg p \vee \neg q) && \text{(distribution)}
\end{aligned}$$

Using this new identity, we can re-write the whole expression as

$$\begin{aligned}(y \wedge \neg x) \vee (x \wedge \neg y) &= (x \wedge (\neg x \vee \neg y)) \vee (y \wedge (\neg x \vee \neg y)) \\ &= (x \vee y) \wedge (\neg x \vee \neg y) && \text{(collect)} \\ &= (x \vee y) \wedge \neg (x \wedge y) && \text{(de Morgan's)}\end{aligned}$$

which gives us the second alternative we are looking for. A later chapter shows how to construct an algorithm to do this sort of simplification mechanically so as to reduce our workload and reduce the chance of error.

1.2 Sets and Functions

The concept of a **set** and the theory behind such objects is a fundamental part of mathematics. Informally, a set is simply a well defined collection of **elements**. Here we mainly deal with sets of numbers, but it is important to note that the elements can be anything you want.

We can define a set using one of several methods. Firstly we can enumerate the elements, writing them down between a pair of braces. For example, one might define the set A of whole numbers between two and eight (inclusive) as

$$A = \{2,3,4,5,6,7,8\}.$$

The **cardinality** of a finite set is the number of elements it contains. For the set A, this is denoted by $|A|$ such that from the example above

$$|A| = 7.$$

If the element a is in the set A, we say a is a **member** of A or write

$$a \in A.$$

The ordering of the elements in the set does not matter, only their membership or non-membership. So we can define another set as

$$B = \{8,7,6,5,4,3,2\}$$

and be safe in the knowledge that $A = B$. However, note that elements cannot occur in a set more than once; a set where repetitions are allowed is sometimes called a **bag** or **multi-set** but is beyond the scope of this discussion.

There are at least two predefined sets which have a clear meaning but are hard to define using any other notation:

Definition 8. The set $\emptyset$, called the **null set** or **empty set**, is the set which contains no elements. Note that $\emptyset$ is a set not an element, one cannot write the empty set as $\{\emptyset\}$ since this is the set with one element, that element being the empty set.

Definition 9. The contents of the set $\mathscr{U}$, called the **universal set**, depends on the context. Roughly speaking, it contains every element from the problem being considered.

As a side note, since the elements in a set can be anything we want they can potentially be other sets. Russell's paradox, discovered by mathematician Bertrand Russell in 1901, is a problem with formal set theory that results from this fact. The paradox is similar to the liar paradox seen earlier and is easily stated by considering A, the set of all sets which do not contain themselves. The question is, does A contain itself ? If it does, it should not be in A by definition but it is; if it does not, it should be in the set A by definition but it is not.

1.2.1 Construction

The above method of definition is fine for small finite sets but when the set is large or even infinite in size, writing down all the elements quickly becomes an unpleasant task ! Where there is a natural sequence to the elements, we write continuation dots to save time and space. For example, the same set A as above might be defined as

$$A = \{2, 3, \ldots, 7, 8\}.$$

This set is finite since there is a beginning and an end; the continuation dots are simply a shortcut. As another example, consider the set of numbers exactly divisible by two that might be defined as

$$C = \{2, 4, 6, 8, \ldots\}.$$

Clearly this set is infinite in size in the sense there is no end to the sequence. In this case the continuation dots are a necessity in order to define the set adequately.

The second way to write down sets is using what is sometimes called set builder notation. Basically speaking, we generate the elements in the set using f, a predicate function:

$$D = \{x : f(x)\}.$$

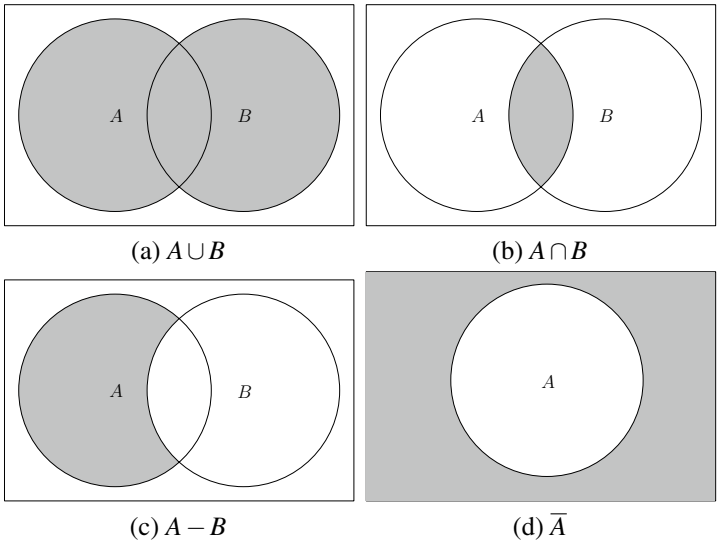
One should read this as "all elements $x \in \mathscr{U}$ such that the predicate $f(x)$ is true". Using set builder notation we can define sets in a more programmatic manner, for example

$$A = \{x : 2 \leq x \leq 8\},$$

and

$$C = \{x : x > 0 \wedge x \equiv 0 \pmod 2\}$$

define the same sets as we explicitly wrote down above.



(a) $A \cup B$ (b) $A \cap B$

(c) $A - B$ (d) $\overline{A}$

Figure 1.1 A collection of Venn diagrams for standard set operations.

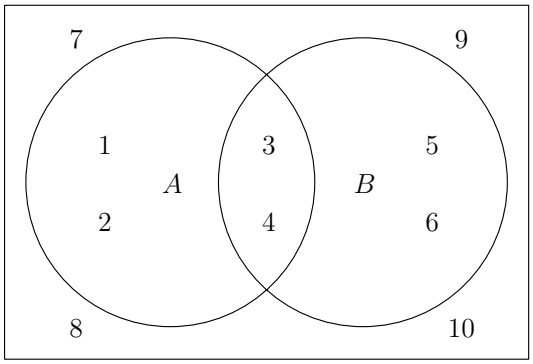


Figure 1.2 An example Venn diagram showing membership of two sets.

1.2.2 Operations

Definition 10. A **sub-set**, say B, of a set A is such that for every $x \in B$ we have that $x \in A$. This is denoted $B \subseteq A$. Conversely, we can say A is a **super-set** of B and write $A \supseteq B$.

Note that every set is a sub-set and super-set of itself and that $A = B$ only if $A \subseteq B$ and $B \subseteq A$. If $A \neq B$, we use the terms **proper sub-set** and **proper super-set** and write $B \subset A$ and $B \supset A$ respectively.

Definition 11. For sets A and B, we have that

- The **union** of A and B is $A \cup B = \{x : x \in A \vee x \in B\}$.
- The **intersection** of A and B is $A \cap B = \{x : x \in A \wedge x \in B\}$.
- The **difference** of A and B is $A - B = \{x : x \in A \wedge x \notin B\}$.
- The **complement** of A is $\overline{A} = \{x : x \in \mathscr{U} \wedge x \notin A\}$.

We say A and B are **disjoint** or **mutually exclusive** if $A \cap B = \emptyset$. Note also that the complement operation can be re-written $A - B = A \cap \overline{B}$.

Definition 12. The power set of a set A, denoted $\mathscr{P}(A)$, is the set of every possible sub-set of A. Note that $\emptyset$ is a member of all power sets.

On first reading, these formal definitions can seem a bit abstract and slightly scary. However, we have another tool at our disposal which describes what they mean in a visual way. This tool is the **Venn diagram**, named after mathematician John Venn who invented the concept in 1881. The basic idea is that sets are represented by regions inside an enclosure that implicitly represents the universal set $\mathscr{U}$. By placing these regions inside each other and overlapping their boundaries, we can describe most set-related concepts very easily.

Figure 1.1 details four Venn diagrams which describe how the union, intersection, difference and complement operations work. The shaded areas of each Venn diagram represent the elements which are in the resulting set. For example, in the diagram for $A \cup B$ the shaded area covers all of the sets A and B: the result contains all elements in either A or B or both. As a simple concrete example, consider the sets

$$A = \{1,2,3,4\}$$
$$B = \{3,4,5,6\}$$

where the universal set is

$$\mathscr{U} = \{1,2,3,4,5,6,7,8,9,10\}.$$

Figure 1.2 shows membership in various settings; recall that those elements within a given region are members of that set. Firstly, we can take the union of A and B as $A \cup B = \{1,2,3,4,5,6\}$ which contains all the elements which are either members of A or B or both. Note that elements 3 and 4 do not appear twice in the result. The intersection of A and B can be calculated as $A \cap B = \{3,4\}$ since these are the elements that are members of both A and B. The difference between A and B, that is the elements in A that are not in B, is $A - B = \{1,2\}$. Finally, the complement of A is all numbers which are not in A, that is $\overline{A} = \{5,6,7,8,9,10\}$.

The union and intersection operations preserve a law of cardinality called the **principle of inclusion** in the sense that we can calculate the cardinality of the output from the cardinality of the inputs as

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

This property is intuitively obvious since those elements in both A and B will be counted twice and hence need subtraction via the last term. We can even check it: in our example above $|A| = 4$ and $|B| = 4$. Checking our results we have that $|A \cup B| = 6$ and $|A \cap B| = 2$ and so by the principle of inclusion we should have $6 = 4 + 4 - 2$ which makes sense.

1.2.3 Numeric Sets

Using this basic notation, we can define three important numeric sets which are used extensively later on in this chapter.

Definition 13. The **integers** are whole numbers which can be positive or negative and also include zero

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}.$$

The **natural numbers** are whole numbers which are positive

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}.$$

The **rational numbers** are those which can be expressed in the form p/q where p and q are integers called the **numerator** and **denominator**

$$\mathbb{Q} = \{p/q : p \in \mathbb{Z} \wedge q \in \mathbb{Z} \wedge q \neq 0\}.$$

Clearly the set of rational numbers is a super-set of both $\mathbb{Z}$ and $\mathbb{N}$ since, for example, we can write $p/1$ to represent any integer p as a member of $\mathbb{Q}$. However, not all numbers are rational. Some are **irrational** in the sense that it is impossible to find a p and q such that they exactly represent the required result; examples include the value of π.

1.2.4 Functions

Definition 14. If A and B are sets, a **function** f from A to B is a process that maps each element of A to an element of B. We write this as

$$f : A \to B$$

where A is termed the **domain** of f and B is the **codomain** of f. For an element $x \in A$, which we term the **preimage**, there is only one $y = f(x) \in B$ which is termed the **image** of x. Finally, the set

$$\{y : y = f(x) \wedge x \in A \wedge y \in B\}$$

which is all possible results, is termed the **range** of f and is always a sub-set of the **codomain**.

As a concrete example, consider a function INV which takes an integer x as input and produces the rational number $1/x$ as output:

$$\text{INV} : \mathbb{Z} \to \mathbb{Q}, \ \ \text{INV}(x) = 1/x.$$

Note that here we write the **function signature** which defines the domain and codomain of INV inline with the definition of the **function behaviour**. In this case the domain of INV is $\mathbb{Z}$ since it takes integers as input; the codomain is $\mathbb{Q}$ since it produces rational numbers as output. If we take an integer and apply the function to get something like $\text{INV}(2) = 1/2$, we have that $1/2$ is the image of 2 or conversely 2 is the preimage of $1/2$ under INV.

From this definition it might seem as though we can only have functions with one input and one output. However, remember that we are perfectly entitled to have sets of sets so we can easily define a function f, for example, as

$$f : A \times A \to B.$$

This function takes elements from the Cartesian product of A as input and produces an element of B as output. So since pairs $(x,y) \in A \times A$ are used as input, f can in some sense accept two input values. As a concrete example consider the function

$$\text{MAX} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \ \ \text{MAX}(x,y) = \begin{cases} x & \text{if} \ \ x > y \\ y & \text{otherwise.} \end{cases}$$

This is the maximum function on integers; it takes two integers as input and produces an integer, the maximum of the inputs, as output. So if we take a pair of integers, say $(2,4)$, and apply the function we get $\text{MAX}(2,4) = 4$ where we usually omit the parentheses around the pair of inputs. In this case, the domain of MAX is $\mathbb{Z} \times \mathbb{Z}$ and the codomain is $\mathbb{Z}$; the integer 4 is the image of the pair $(2,4)$ under MAX.

Definition 15. For a given function f, we say that f is

- **surjective** if the range equals the codomain, i.e., there are no elements in the codomain which do not have a preimage in the domain.
- **injective** if no two elements in the domain have the same image in the range.
- **bijective** if the function is both surjective and injective, i.e., every element in the domain is mapped to exactly one element in the codomain.

Using the examples above, we clearly have that INV is not surjective but MAX is. This follows because we can construct a rational $2/3$ which does not have an integer preimage under INV so the function cannot be surjective. Equally, for any integer x in the range of MAX there is always a pair (x,y) in the domain such that $x > y$ so

MAX is surjective, in fact there are lots of them since $\mathbb{Z}$ is infinite in size ! In the same way, we have that INV is injective but MAX is not. Only one preimage x maps to the value $1/x$ in the range under INV but there are multiple pairs (x,y) which map to the same image under MAX, for example 4 is the image of both $(1,4)$ and $(2,4)$ under MAX.

Definition 16. Given two functions $f : A \to B$ and $g : B \to C$, the **composition** of f and g is denoted

$$g \circ f : A \to C.$$

Given some input $x \in A$, this composition is equivalent to applying $y = f(x)$ and then $z = g(y)$ to get the result $z \in C$. More formally, we have

$$(g \circ f)(x) = g(f(x)).$$

The notation $g \circ f$ should be read as "apply g to the result of applying f".

Definition 17. The **identity function** $\mathscr{I}$ on a set A is defined by

$$\mathscr{I} : A \to A, \;\; \mathscr{I}(x) = x$$

so that it maps all elements to themselves. Given two functions f and g defined by $f : A \to B$ and $g : B \to A$, if $g \circ f$ is the identity function on set A and $f \circ g$ is the identity on set B, then f is the **inverse** of g and g is the inverse of f. We denote this by $f = g^{-1}$ and $g = f^{-1}$. If a function f has an inverse, we hence have $f^{-1} \circ f = \mathscr{I}$.

The inverse of a function maps elements from the codomain back into the domain, essentially reversing the original function. It is easy to see not all functions have an inverse. For example, if the function is not injective then there will be more than one potential preimage for the inverse of any image.

At first glance, it might seem like our example functions both have inverses but they do not. For example, given some value $1/x$, we can certainly find x but we have already said that numbers like $2/3$ also exist in the codomain so we cannot invert all the values we might come across. However, consider the example of a successor function on integers

$$\text{SUCC} : \mathbb{Z} \to \mathbb{Z}, \;\; \text{SUCC}(x) = x + 1$$

which takes an integer x as input and produces $x+1$ as output. The function is bijective since the codomain and range are the same and no two integers have the same successor. Thus we have an inverse and it is easy to describe as

$$\text{PRED} : \mathbb{Z} \to \mathbb{Z}, \;\; \text{PRED}(x) = x - 1$$

which is the predecessor function: it takes an integer x as input and produces $x-1$ as output. To see that $\text{SUCC}^{-1} = \text{PRED}$ and $\text{SUCC}^{-1} = \text{PRED}$ note that

$$(\text{PRED} \circ \text{SUCC})(x) = (x+1) - 1 = x$$

which is the identity function. Conversely,

$$(\text{SUCC} \circ \text{PRED})(x) = (x-1)+1 = x$$

which is also the identity function.

1.2.5 Relations

Definition 18. We call a sequence of n elements $(x_0, x_1, \ldots, x_{n-1})$ an **n-tuple** or simply a **tuple** when the number of elements is irrelevant. In the specific case of $n = 2$, we call (x_0, x_1) a **pair**. The i-th element of the tuple x is denoted x_i, and the number of elements in a particular tuple x may be written as $|x|$.

Definition 19. The **Cartesian product** of n sets, say $A_0, A_1, \ldots, A_{n-1}$, is defined as

$$A_0 \times A_1 \times \cdots \times A_{n-1} = \{(a_0, a_1, \ldots, a_{n-1}) : a_0 \in A_0, a_1 \in A_1, \ldots, a_{n-1} \in A_{n-1}\}.$$

In the most simple case of $n = 2$, the Cartesian product $A_0 \times A_1$ is the set of all possible pairs where the first item in the pair is a member of A_0 and the second item is a member of A_1.

The Cartesian product of a set A with itself n times is denoted A^n. To be complete, we define $A^0 = \emptyset$ and $A^1 = A$. Finally, by writing A^* we mean the Cartesian product of A with itself a finite number of times.

Cartesian products are useful to us since they allow easy description of sequences, or **vectors**, of elements. Firstly, we often use them to describe vectors of elements from a set. So for example, say we have the set of digits $A = \{0, 1\}$. The Cartesian product of A with itself is

$$A \times A = \{(0,0), (0,1), (1,0), (1,1)\}.$$

If you think of the pairs as more generally being vectors of these digits, the Cartesian product A^n is the set of all possible vectors of 0 and 1 which are n elements long.

Definition 20. Informally, a **binary relation** f on a set A is like a predicate function which takes members of the set as input and "filters" them to produce an output. As a result, for a set A the relation f forms a sub-set of $A \times A$. For a given set A and a binary relation f, we say f is

- **reflexive** if $f(x,x) =$ true for all $x \in A$.
- **symmetric** if $f(x,y) =$ true implies $f(y,x) =$ true for all $x, y \in A$.
- **transitive** if $f(x,y) =$ true and $f(y,z) =$ true implies $f(x,z) =$ true for all $x, y, z \in A$.

If f is reflexive, symmetric and transitive, then we call it an **equivalence relation**.

The easiest way to think about this is to consider a concrete example such as the case where our set is $A = \{1,2,3,4\}$ such that the Cartesian product is

$$A \times A = \left\{ \begin{array}{l} (1,1),\ (1,2),\ (1,3),\ (1,4), \\ (2,1),\ (2,2),\ (2,3),\ (2,4), \\ (3,1),\ (3,2),\ (3,3),\ (3,4), \\ (4,1),\ (4,2),\ (4,3),\ (4,4) \end{array} \right\}.$$

Imagine we define a function

$$\text{EQU} : \mathbb{Z} \times \mathbb{Z} \rightarrow \{\text{true}, \text{false}\}, \text{EQU}(x,y) = \left\{ \begin{array}{ll} \text{true} & \text{if } \ x = y \\ \text{false} & \text{otherwise} \end{array} \right.$$

which tests whether two inputs are equal. Using the function we can form a sub-set of $A \times A$ (called A_{EQU} for example) in the sense that we can pick out the pairs (x,y)

$$A_{\text{EQU}} = \{(1,1),(2,2),(3,3),(4,4)\}$$

where $\text{EQU}(x,y) = \text{true}$. For members of A, say $x,y,z \in A$, we have that $\text{EQU}(x,x) = \text{true}$ so the relation is reflexive. If $\text{EQU}(x,y) = \text{true}$, then $\text{EQU}(y,x) = \text{true}$ so the relation is also symmetric. Finally, if $\text{EQU}(x,y) = \text{true}$ and $\text{EQU}(y,z) = \text{true}$, then we must have that $\text{EQU}(x,z) = \text{true}$ so the relation is also transitive and hence an equivalence relation.

Now imagine we define another function

$$\text{LTH} : \mathbb{Z} \times \mathbb{Z} \rightarrow \{\text{true}, \text{false}\}, \text{LTH}(x,y) = \left\{ \begin{array}{ll} \text{true} & \text{if } \ x < y \\ \text{false} & \text{otherwise} \end{array} \right.$$

which tests whether one input is less than another, and consider the sub-set of A

$$A_{\text{LTH}} = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$

of all pairs (x,y) with $x,y \in A$ such that $\text{LTH}(x,y) = \text{true}$. It cannot be that $\text{LTH}(x,x) = \text{true}$, so the relation is not reflexive; we say it is irreflexive. It also cannot be that if $\text{LTH}(x,y) = \text{true}$ then $\text{LTH}(y,x) = \text{true}$, so the relation is also not symmetric; it is anti-symmetric. However, if both $\text{LTH}(x,y) = \text{true}$ and $\text{LTH}(y,z) = \text{true}$, then $\text{LTH}(x,z) = \text{true}$ so the relation is transitive.

1.3 Boolean Algebra

Most people are taught about simple numeric algebra in school. One has

- a set of values, say $\mathbb{Z}$,
- a list of operations, say $-$, $+$ and $\cdot$,
- and a list of axioms which dictate how $-$, $+$ and $\cdot$ work.

You might not know the names for these axioms but you probably know *how* they work, for example you probably know for some $x, y, z \in \mathbb{Z}$ that $x + (y + z) = (x + y) + z$.

In the early 1840s, mathematician George Boole combined propositional logic and set theory to form a system which we now call Boolean algebra [4]. Put (rather too) simply, Boole saw that working with a logic expression x is much the same as working with a number y. One has

- a set of values, $\{\text{false}, \text{true}\}$,
- a list of operations $\neg$, $\vee$ and $\wedge$,
- and a list of axioms that dictate how $\neg$, $\vee$ and $\wedge$ work.

Ironically, at the time his work was viewed as somewhat obscure and indeed Boole himself did not necessarily regard logic directly as a mathematical concept. However, the step of finding common themes within two such concepts and unifying them into one has been a powerful tool in mathematics, allowing more general thinking about seemingly diverse objects. It was not until 1937 that Claude Shannon, then a student of both electrical engineering and mathematics, saw the potential of using Boolean algebra to represent and manipulate digital information [60].

Definition 21. A **binary operation** $\odot$ on some set A is a function

$$\odot : A \times A \to A$$

while a **unary operation** $\oslash$ on A is a function

$$\oslash : A \to A.$$

Definition 22. Define a set $\mathbb{B} = \{0, 1\}$ on which there are two binary operators $\wedge$ and $\vee$ and a unary operator $\neg$. The members of $\mathbb{B}$ act as **identity elements** for $\wedge$ and $\vee$. These operators and identities are governed by a number of axioms which should look familiar:

equivalence	$x \leftrightarrow y$	$\equiv (x \rightarrow y) \wedge (y \rightarrow x)$
implication	$x \rightarrow y$	$\equiv \neg x \vee y$
involution	$\neg\neg x$	$\equiv x$
idempotency	$x \wedge x$	$\equiv x$
commutativity	$x \wedge y$	$\equiv y \wedge x$
association	$(x \wedge y) \wedge z$	$\equiv x \wedge (y \wedge z)$
distribution	$x \wedge (y \vee z)$	$\equiv (x \wedge y) \vee (x \wedge z)$
de Morgan's	$\neg(x \wedge y)$	$\equiv \neg x \vee \neg y$
identity	$x \wedge 1$	$\equiv x$
null	$x \wedge 0$	$\equiv 0$
inverse	$x \wedge \neg x$	$\equiv 0$
absorption	$x \wedge (x \vee y)$	$\equiv x$
idempotency	$x \vee x$	$\equiv x$
commutativity	$x \vee y$	$\equiv y \vee x$
association	$(x \vee y) \vee z$	$\equiv x \vee (y \vee z)$
distribution	$x \vee (y \wedge z)$	$\equiv (x \vee y) \wedge (x \vee z)$
de Morgan's	$\neg(x \vee y)$	$\equiv \neg x \wedge \neg y$
identity	$x \vee 0$	$\equiv x$
null	$x \vee 1$	$\equiv 1$
inverse	$x \vee \neg x$	$\equiv 1$
absorption	$x \vee (x \wedge y)$	$\equiv x$

We call the $\wedge$, $\vee$ and $\neg$ operators NOT, AND and OR respectively.

Notice how this description of a Boolean algebra unifies the concepts of propositional logic and set theory: 0 and false and $\emptyset$ are sort of equivalent, as are 1 and true and $\mathscr{U}$. Likewise, $x \wedge y$ and $A \cap B$ are sort of equivalent, as are $x \vee y$ and $A \cup B$ and $\neg x$ and $\overline{A}$: you can even draw Venn diagrams to illustrate this fact. It might seem weird to see the statement

$$x \vee x = x$$

written down but we know what it means because the axioms allow one to manipulate the statement as follows:

$$\begin{aligned} x \vee x &= (x \vee x) \wedge 1 && \text{(identity)} \\ &= (x \vee x) \wedge (x \vee \neg x) && \text{(inverse)} \\ &= x \vee (x \wedge \neg x) && \text{(distribution)} \\ &= x \vee 0 && \text{(inverse)} \\ &= x && \text{(identity)} \end{aligned}$$

Also notice that the $\wedge$ and $\vee$ operations in a Boolean algebra behave in a similar way to $\cdot$ and $+$ in a numerical algebra; for example we have that $x \vee 0 = x$ and $x \wedge 0 = 0$. As such, $\wedge$ and $\vee$ are often termed (and written as) the "product" and "sum" operations.

1.3.1 Boolean Functions

Definition 23. In a given Boolean algebra over the set $\mathbb{B}$, a Boolean function f can be described as

$$f : \mathbb{B}^n \to \mathbb{B}.$$

That is, it takes an n-tuple of elements from the set as input and produces an element of the set as output.

This definition is more or less the same as the one we encountered earlier. For example, consider a function f that takes two inputs whose signature we can write as

$$f : \mathbb{B}^2 \to \mathbb{B}$$

so that for $x, y, z \in \mathbb{B}$, the input is a pair (x, y) and the output for a given x and y is written $z = f(x, y)$. We can define the actual mapping performed by f in two ways. Firstly, and similarly to the truth tables seen previously, we can simply enumerate the mapping between inputs and outputs:

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

However, with a large number of inputs this quickly becomes difficult so we can also write f as a Boolean expression

$$f : \mathbb{B}^2 \to \mathbb{B}, \quad f(x,y) = (\neg x \wedge y) \vee (x \wedge \neg y)$$

which describes the operation in a more compact way.

If we require more than one output, say m outputs, from a given function, we can think of this just m separate functions each of which produces one output from the same inputs. For example, taking $m = 2$ a function with the signature

$$g : \mathbb{B}^2 \to \mathbb{B}^2$$

can be split into two functions

$$g_0 : \mathbb{B}^2 \to \mathbb{B}$$
$$g_1 : \mathbb{B}^2 \to \mathbb{B}$$

so that the outputs of g_0 and g_1 are grouped together to form the output of g. We can again specify the functions either with a truth table

x	y	$g_0(x,y)$	$g_1(x,y)$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

or a pair of expressions

$$g_0 : \mathbb{B}^2 \to \mathbb{B}, \;\; g_0(x,y) = (\neg x \wedge y) \vee (x \wedge \neg y)$$
$$g_1 : \mathbb{B}^2 \to \mathbb{B}, \;\; g_1(x,y) = (x \wedge y)$$

such that the original function g can be roughly described as

$$g(x,y) = (g_0(x,y), g_1(x,y)).$$

This end result is often termed a **vectorial** Boolean function: the inputs and outputs are vectors over the set $\mathbb{B}$ rather than single elements of the set.

1.3.2 Normal Forms

Definition 24. Consider a Boolean function f with n inputs, for example $f(x_0, x_1, \ldots, x_{n-1})$.

When the expression for f is written as a sum (i.e., OR) of terms which each comprise the product (i.e., AND) of a number of inputs, it is said to be in **disjunctive normal form** or **Sum of Products (SoP)** form; the terms in this expression are called the **minterms**. Conversely, when the expression for f is written as a product (i.e., AND) of terms which each comprise the sum (i.e., OR) of a number of inputs, it is said to be in **conjunctive normal form** or **Product of Sums (PoS)** form; the terms in this expression are called the **maxterms**.

This is a formal definition for something very simple to describe by example. Consider a function f which is described by

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

The minterms are the second and third rows of this table, the maxterms are the first and fourth lines. An expression for f in SoP form is

$$f_{SoP}(x,y) = (\neg x \wedge y) \vee (x \wedge \neg y).$$

Notice how the terms $\neg x \wedge y$ and $x \wedge \neg y$ are the minterms of f: when the term is 1 or 0, the corresponding minterm is 1 or 0. It is usually crucial that all the variables

appear in all the minterms so that the function is exactly described. To see why this is so, consider writing an incorrect SoP expression by removing the reference to y from the first minterm so as to get

$$(\neg x) \vee (x \wedge \neg y).$$

Since $\neg x$ is 1 for the first and second rows, rather than just the second as was the case with $\neg x \wedge y$, we have described another function $f' \neq f$ with the truth table

x	y	$f'(x,y)$
0	0	1
0	1	1
1	0	1
1	1	0

In a similar way to above, we can construct a PoS expression for f as

$$f_{PoS}(x,y) = (x \vee y) \wedge (\neg x \vee \neg y).$$

In this case the terms $x \vee y$ and $\neg x \vee \neg y$ are the maxterms. To verify that the two forms represent the same function, we can apply de Morgan's axiom to get

$$\begin{aligned} f_{PoS}(x,y) &= (x \vee y) \wedge (\neg x \vee \neg y) \\ &= (\neg x \wedge \neg y) \vee (x \wedge y) \text{ (de Morgan's).} \end{aligned}$$

Although this might not seem particularly interesting at first glance, it gives us quite a powerful technique. Given a truth table for an arbitrary function, we can construct an expression for the function simply by extracting either the minterms or maxterms and constructing standard SoP or PoS forms.

1.4 Number Systems

Definition 25. One can write an integer x as

$$x = \pm \sum_{i=0}^{n-1} x_i \cdot b^i$$

where each x_i is taken from the **digit set** $\mathbb{D} = \{0 \ldots b-1\}$. This is called the base-b **expansion** of x; the selection of b determines the **base** or **radix** of the number system x is written in. The same integer can be written as

$$x = \pm (x_0, x_1, \ldots, x_{n-1})^{(b)}$$

which is called the base-b vector representation of x.

Definition 26. We use $x_{(b)}$ to denote the integer literal x in base-b. Where it is not clear, we use $|x_{(b)}|$ to denote n, the number of digits in the base-b expansion of x. At times this might look a little verbose or awkward, however it allows us to be exact at all times about the meaning of what we are writing.

We are used to working in the **decimal** or **denary** number system by setting $b = 10$, probably because we (mostly) have ten fingers and toes. In this case, we have that $x_i \in \{0 \ldots 9\}$ with each digit representing a successive power of ten. Using this number system we can easily write down integers such as

$$123_{(10)} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

Note that for all y, we define $y^0 = 1$ and $y^1 = y$ so that in the above example $10^0 = 1$ and $10^1 = 10$. Furthermore, note that we can also have negative values of i so, for example, $10^{-1} = 1/10$ and $10^{-2} = 1/100$. Using this fact, we can also write down rational numbers such as

$$123.3125_{(10)} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 3 \cdot 10^{-1} + 1 \cdot 10^{-2} + 2 \cdot 10^{-3} + 5 \cdot 10^{-4}.$$

Of course, one can select other values for b to represent x in different bases. Common choices are $b = 2$, $b = 8$ and $b = 16$ which form the **binary**, **octal** and **hexadecimal** number systems. Irrespective of the base, integer and rational numbers are written using exactly the same method so that

$$\begin{aligned} 123_{(10)} &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1111011_{(2)} \end{aligned}$$

and

$$\begin{aligned} 123.3125_{(10)} &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-2} + 1 \cdot 2^{-4} \\ &= 1111011.0101_{(2)}. \end{aligned}$$

The **decimal point** we are used to seeing is termed the **binary point** in this case. Considering rational numbers, depending on the base used there exist some values which are not exactly representable. Informally, this is obvious since when we write $1/3$ as a decimal number we have $0.3333r_{(10)}$ with the appended letter r meaning that the sequence recurs infinitely. Another example is $1/10$ which can be written as $0.1_{(10)}$ but is not exactly representable in binary, the closest approximation being $0.000110011r_{(2)}$.

There are several properties of the binary expansion of numbers that warrant a brief introduction. Firstly, the number of digits set to 1 in a binary expansion of some value x is called the **Hamming weight** of x after Richard Hamming, a researcher at Bell Labs. Hamming also introduced the idea of the distance between two values which captures how many digits differ; this is named the **Hamming distance**. For example, consider the two values

$$\begin{aligned} x &= 9_{(10)} = 1\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = 1001_{(2)} \\ y &= 14_{(10)} = 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 1110_{(2)}. \end{aligned}$$

The Hamming weights of x and y, say $H(x)$ and $H(y)$, are 2 and 3 respectively; the Hamming distance between x and y is 3 since three bits differ.

Using the hexadecimal numbers system is slightly complicated by the fact we are used to only using the ten decimal digits $0\ldots9$ yet need to cope with sixteen hexadecimal digits. As a result, we use the letters $A\ldots F$ to write down the base-10 numbers $10\ldots15$. This means we have

$$\begin{aligned} 7B_{(16)} &= 7\cdot 16^1 + B\cdot 16^0 \\ &= 7\cdot 16^1 + 11\cdot 16^0 \\ &= 123_{(10)} \quad . \end{aligned}$$

Each octal or hexadecimal digit represents exactly three or four binary digits respectively; using this shorthand one can more easily write and remember long vectors of binary digits.

Aside from the issue of not being able to exactly represent some rational numbers in some bases, it is crucial to realise that an actual number x and how we represent it are largely independent. For example we can write any of

$$\begin{aligned} 123_{(10)} &= 1111011_{(2)} \\ &= 173_{(8)} \\ &= 7B_{(16)} \end{aligned}$$

yet they all represent the same number, they still mean the same thing, exactly the same rules apply in all bases even if they seem unnatural or unfamiliar. The choice of base then is largely one of convenience in a given situation, so we really want to select a base suitable for use in the context of digital computers and computation.

1.4.1 Converting Between Bases

Considering just integers for the moment, converting between different bases is a fairly simple task. Algorithm 1.1 demonstrates how it can be achieved; clearly we need to be able to do arithmetic in both bases for it to be practical. Simply put, we extract one digit of the result at a time. As an example run, consider converting $123_{(10)}$ into binary, i.e., $x = 123$, $\hat{b} = 10$ and $\bar{b} = 2$:

Input: An integer number x represented in base-$\hat{b}$.
Output: The same number represented in base-$\bar{b}$.

```
1  INTEGER-BASE-CONVERT(x, b̂, b̄)
2      i ← 0
3      t ← x
4      if x < 0 then
5          t ← −t
6      r ← 0
7      while t ≠ 0 do
8          m ← t mod b̄
9          t ← ⌊t/b̄⌋
10         r ← r + (m · b̄^i)
11         i ← i + 1
12     if x < 0 then
13         r ← −r
14     return r
15 end
```

Algorithm 1.1: An algorithm for converting integers from representation in one base to another.

Input: A rational number x represented in base-$\hat{b}$, an integer d determining the number of digits required.
Output: The same number represented in base-$\bar{b}$.

```
1  FRACTION-BASE-CONVERT(x, b̂, b̄, d)
2      i ← 1
3      t ← x
4      if x < 0 then
5          t ← −t
6      r ← 0
7      while t ≠ 0 and i < d do
8          m ← t · b̄
9          n ← ⌊m⌋
10         r ← r + (n · b̄^−(i+1))
11         t ← m − n
12         i ← i + 1
13     return r
14 end
```

Algorithm 1.2: An algorithm for converting the fractional part of a rational number from one base to another.

i	m	t	r
–	–	$123_{(10)}$	$0000000_{(2)} = 0$
0	1	$61_{(10)}$	$0000001_{(2)} = 2^0$
1	1	$30_{(10)}$	$0000011_{(2)} = 2^0 + 2^1$
2	0	$15_{(10)}$	$0000011_{(2)} = 2^0 + 2^1$
3	1	$7_{(10)}$	$0001011_{(2)} = 2^0 + 2^1 + 2^3$
4	1	$3_{(10)}$	$0011011_{(2)} = 2^0 + 2^1 + 2^3 + 2^4$
5	1	$1_{(10)}$	$1111011_{(2)} = 2^0 + 2^1 + 2^3 + 2^4 + 2^5$
6	1	$0_{(10)}$	$1111011_{(2)} = 2^0 + 2^1 + 2^3 + 2^4 + 2^5 + 2^6$

One can see how the result is built up a digit at a time by finding the remainder of x after division by the target base. Finally, if the input was negative we negate the accumulated result and return it.

Dealing with rational numbers is more tricky. Clearly we can still convert the integer part of such a number using Algorithm 1.1. The problem with using a similar approach with the fractional part is that if the number cannot be represented exactly in a given base, our loop to generate digits will never terminate. To cope with this Algorithm 1.2 takes an extra parameter which determines the number of digits we require. As another example, consider the conversion of $0.3125_{(10)}$ into four binary digits, i.e., $x = 0.3125$, $\hat{b} = 10$, $\bar{b} = 2$ and $d = 4$:

i	n	t	r
–	–	$0.3125_{(10)}$	$0.0000_{(2)} = 0$
1	1	$0.6250_{(10)}$	$0.0100_{(2)} = 2^{-2}$
2	0	$0.2500_{(10)}$	$0.0100_{(2)} = 2^{-2}$
3	1	$0.5000_{(10)}$	$0.0101_{(2)} = 2^{-2} + 2^{-4}$

On each step, we move one digit across to the left of the fractional point by multiplying by the target base. This digit is extracted using the floor function; $\lfloor x \rfloor$ denotes the integer part of a rational number x. Then, so we can continue processing the rest of the number, we remove the digit to the left of the fractional point by subtracting it from our working accumulator. We already set the sign of our number when converting the integer part of the input so we do not need to worry about that.

1.4.2 Bits, Bytes and Words

Concentrating on binary numbers, a single binary digit is usually termed a **bit**. To represent numbers we group together bits into larger sequences; we usually term these **binary vectors**.

Definition 27. An n-bit **binary vector** or **bit-vector** is a member of the set $\mathbb{B}^n$ where $\mathbb{B} = \{0, 1\}$, i.e., it is an n-tuple of bits. We use x_i to denote the i-th bit of the binary

vector x and $|x| = n$ to denote the number of bits in x. Instead of writing out $x \in \mathbb{B}^n$ fully as the n-tuple $(x_0, x_1, \ldots, x_{n-1})$ we typically just list the digits in sequence.

For example, consider the following bit-vector

$$(1,1,0,1,1,1,1)$$

which can be less formally written as

$$1111011$$

and has seven bits in it. Note that there is no notation for what base the number is in and so on: this is not a mistake. This is just a vector of binary digits, in one context it could be representing a number but in another context perhaps it represents part of an image: there is a clear distinction between a number x and the representation of x as a bit-vector.

Definition 28. We use the **concatenation** operator (written as $x : y$) to join together two or more bit-vectors; for example $111 : 1011$ describes the bit-vector 1111011.

We do however need some standard way to make sure we know which bits in the vector represent which bits from the more formal n-tuple; this is called an **endianness** convention. Usually, and as above, we use a **little-endian** convention by reading the bits from right to left, giving each bit is given an index. So writing the number $1111011_{(2)}$ as the vector 1111011 we have that

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 1 \\ x_2 &= 0 \\ x_3 &= 1 \\ x_4 &= 1 \\ x_5 &= 1 \\ x_6 &= 1 \end{aligned}$$

If we interpret this vector as a binary number, it therefore represents $1111011_{(2)} = 123_{(10)}$. In this case, the right-most bit (bit zero or x_0) is termed the **least-significant** while the left-most bit (bit $n-1$ or x_{n-1}) is the **most-significant**. However, there is no reason why little-endian this is the only option: a **big-endian** naming convention reverses the indices so we now read them left to right. The left-most bit (bit zero or x_0) is now the most-significant and the right-most (bit $n-1$ or x_{n-1}) is the least-significant. If the same vector is interpreted in big-endian notation, we find that

$$
\begin{aligned}
x_0 &= 1 \\
x_1 &= 1 \\
x_2 &= 1 \\
x_3 &= 1 \\
x_4 &= 0 \\
x_5 &= 1 \\
x_6 &= 1
\end{aligned}
$$

and if interpreted as a binary number, the value now represents $1101111_{(2)} = 111_{(10)}$.

We have special terms for certain length vectors. A vector of eight bits is termed a **byte** (or octet) while a vector of four bits is a **nibble** (or nybble). A given context will, in some sense, have a natural value for n. Vectors of bits which are of this natural size are commonly termed **words**; one may also see **half-word** and **double-word** used to describe $(n/2)$-bit and $2n$-bit vectors. The word size is a somewhat vague concept but becomes clear with some context. For example, when you hear a computer processor described as being 32-bit or 64-bit this is, roughly speaking, specifying that the word size is 32 or 64 bits. Nowadays, the word size of a processor is normally a power-of-two since this simplifies a number of issues. But this was not always the case: the EDSAC computer used a 17-bit word since this represented a limit in terms of expenditure on hardware !

The same issues of endianness need consideration in the context of bytes and words. For example, one can regard a 32-bit word as consisting of four 8-bit bytes. Taking W, X, Y and Z as bytes, we can see that the word can be constructed two ways. Using a little-endian approach, Z is byte zero, Y is byte one, X is byte two and W is byte three; we have the word constructed as $W : X : Y : Z$. Using a big-endian approach the order is swapped so that Z is byte three, Y is byte two, X is byte one and W is byte zero; we have the word constructed as $Z : Y : X : W$.

1.4.3 Representing Numbers and Characters

The sets $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{Q}$ are infinite in size. But so far we have simply written down such numbers; our challenge in using them on a computer is to represent the numbers using finite resources. Most obviously, these finite resources place a bound on the number of digits we can have in our base-b expansion.

We have already mentioned that modern digital computers prefer to work in binary. More specifically, we need to represent a given number within n bits: this includes the sign and the magnitude which combine to form the value. In discussing how this is achieved we use the specific case of $n = 8$ by representing given numbers within a byte.

Decimal	BCD Encoding
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 1.1 A table showing a standard BCD encoding system.

1.4.3.1 Representing Integer Numbers

The **Binary Coded Decimal (BCD)** system is one way of encoding decimal digits as binary vectors. Believe it or not, there are several standards for BCD. The most simple of these is called Simple Binary Coded Decimal (SBCD) or BCD 8421: one represents a single decimal digit as a vector of four binary digits according to Table 1.1. So for example, the number $123_{(10)}$ would be represented (with spacing for clarity) by the vector

$$0001\ \ 0010\ \ 0011.$$

Clearly there are some problems with BCD, most notably that there are six unused encodings, $1010 \ldots 1111$, and hence the resulting representation is not very compact. Also, we need something extra at the start of the encoding to tell us if the number is positive or negative. Even so, there are applications for this sort of scheme where decimal arithmetic is de rigueur.

Consigning BCD to the dustbin of history, we can take a much easier approach to representing unsigned integers. We simply let an n-bit binary vector represent our integer as we have already been doing so far. In such a representation an integer x can take the values

$$0 \leq x \leq 2^n - 1.$$

However, storing signed integers is a bit more tricky since we need some space to represent the sign of the number. There are two common ways of achieving this: the sign-magnitude and twos-complement methods.

The **sign-magnitude** method represents a signed integer in n bits by allocating one bit to store the sign, typically the most-significant, and $n-1$ to store the magnitude. The sign bit is set to one for a negative integer and zero for a positive integer so that x is expressed as

$$x = -1^{x_{n-1}} \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i.$$

Hence we find that our example is represented in eight bits by

$$01111011 = +1 \cdot (2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0) = +123_{(10)}$$

while the negation is represented by

$$11111011 = -1 \cdot (2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0) = -123_{(10)}.$$

In such an n-bit sign-magnitude representation, an integer x can take the values

$$-2^{n-1} - 1 \leq x \leq +2^{n-1} - 1.$$

For our 8-bit example, this means we can represent $-127 \ldots +127$.

Note that in a sign-magnitude representation there are two versions of zero, namely ± 0. This can make constructing arithmetic a little more tricky than it might otherwise be. As alternative to this approach, the **twos-complement** representation removes the need for a dedicated sign bit and thus the problem of there being two versions of zero. Essentially, twos-complement defines the most-significant bit of an n-bit value to represent -2^{n-1} rather than $+2^{n-1}$. Thus, an integer x is written as

$$x = x_{n-1} \cdot -2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i.$$

Starting with the value

$$+123_{(10)} = 01111011$$

we obtain the twos-complement negation as

$$-123_{(10)} = 10000101$$

since

$$-123_{(10)} = 1 \cdot -2^7 + 1 \cdot 2^2 + 1 \cdot 2^0.$$

The range of values in an n-bit twos-complement representation is slightly different than using sign-magnitude; an integer x can represent the values in the range

$$-2^{n-1} \leq x \leq +2^{n-1} - 1.$$

So for our 8-bit example, this means values in the range $-128 \ldots +127$ with the difference from the sign-magnitude version coming as a result of there only being one version of zero: the twos-complement of zero is zero. As you might expect, calculating the twos-complement negation of a number can be done with a simple algorithm; we see how this works when we introduce methods for performing arithmetic with numbers in Chapter 7.

It is crucial to realise that the representation of a number in either method is dependent on the value of n we need to fit them in. For example, setting $n = 4$ the sign-magnitude representation of $-2_{(10)}$ is 1010 while with $n = 8$ it is 10000010. A

similar argument is true of twos-complement: it is not a case of just padding zeros onto one end to make it the right size.

1.4.3.2 Representing Characters

Imagine that we want to represent a sequence or **string** of letters or **characters**; perhaps as used in a call to the `printf` function in a C program. We have already seen how to represent integers using binary vectors that are suitable for digital computers, the question is how can we do the same with characters ? Of course, this is easy, we just need to translate from one to the other. More specifically, we need two functions: $\text{ORD}(x)$ which takes a character x and gives us back the corresponding integer representation, and $\text{CHR}(y)$ which takes an integer representation y and gives us back the corresponding character. But how do we decide how the functions should work ?

Fortunately, people have thought about this and provided standards we can use. One of the oldest, from circa 1967, and most simple is the **American Standard Code for Information Interchange (ASCII)**, pronounced "ass key". ASCII has a rich history but was originally developed to communicate between early "teleprinter" devices. These were like the combination of a typewriter and a telephone: the devices were able to communicate text to each other, before innovations such as the fax machine. Later, but long before monitors and graphics cards existed, similar devices allowed users to send input to early computers and receive output from them. Table 1.2 shows the 128-entry ASCII table which tells us how characters are represented as numbers; 95 are printable characters that we can recognise (including *SPC* which is short for "space"), 33 are non-printable control characters. Originally these would have been used to control the teleprinter rather than to have it print something. For example, the *CR* and *LF* characters (short for "carriage return" and "line feed") would combine to move the print head onto the next line; we still use these characters to mark the end of lines in text files. Other control characters still play a role in more modern computers as well: the *BEL* (short for "bell") characters play a "ding" sound when printed to most UNIX terminals, we have keyboards with keys that relate to *DEL* and *ESC* (short for "delete" and "escape") and so on.

Given the ASCII table, we can see that for example $\text{CHR}(104) = \text{'h'}$, i.e., if we see the integer 104 then this represents the character 'h'. Conversely we have that $\text{ORD}(\text{'h'}) = 104$. Although in a sense any consistent translation between characters and numbers would do, ASCII has some useful properties. Look specifically at the alphabetic characters:

- Imagine we want to convert a character x from lower case into upper case. The lower case characters are represented numerically as the contiguous range $97 \ldots 122$; the upper case characters as the contiguous range $65 \ldots 90$. So we can convert from lower case into upper case simply by subtracting 32. For example:

$$\text{CHR}(\text{ORD}(\text{'a'}) - 32) = \text{'A'}.$$

y ORD(x)	CHR(y) x	y ORD(x)	CHR(y) x	y ORD(x)	CHR(y) x	y ORD(x)	CHR(y) x
0	*NUL*	1	*SOH*	2	*STX*	3	*ETX*
4	*EOT*	5	*ENQ*	6	*ACK*	7	*BEL*
8	*BS*	9	*HT*	10	*LF*	11	*VT*
12	*FF*	13	*CR*	14	*SO*	15	*SI*
16	*DLE*	17	*DC*1	18	*DC*2	19	*DC*3
20	*DC*4	21	*NAK*	22	*SYN*	23	*ETB*
24	*CAN*	25	*EM*	26	*SUB*	27	*ESC*
28	*FS*	29	*GS*	30	*RS*	31	*US*
32	*SPC*	33	!	34	"	35	#
36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~	127	*DEL*

Table 1.2 A table describing the ASCII character encoding.

- Imagine we want to test if one character x is alphabetically before some other y (ignoring the issue of case). The way the ASCII translation is specified, we can simply compare their numeric representation: if we find $\textsc{Ord}(x) < \textsc{Ord}(y)$, then the character x is before the character y in the alphabet.

The ASCII table only has 128 entries: one represents each character with a 7-bit binary vector, or an integer in the range $0 \ldots 128$. However, given that an 8-bit byte is a more convenient size, what about characters that are assigned to the range $128 \ldots 255$? Strict ASCII uses the eighth bit for error correction, but some systems ignore this and define a so-called "extended" ASCII table: characters within it are used for special purposes on a per-computer basis: one computer might give them one meaning, another computer might give them another meaning. For example, characters for specific languages are often defined in this range (e.g., é or ø), and "block" characters are included and used by artists who form text-based pictures.

The main problem with ASCII in a modern context is the lack of spare space in the encoding: it only includes characters suitable for English, so if additional language specific characters are required it cannot scale easily. Although some extensions of ASCII have been proposed, a new standard for extended character sets called **unicode** is now more prevalent.

1.4.3.3 Representing Rational Numbers

The most obvious way to represent rational numbers is simply to allocate a number of bits to the fractional part and the rest to the whole part. For example, given that we have

$$123.3125_{(10)} = 1111011.0101_{(2)},$$

we could simply have an 11-bit value $11110110101_{(2)}$ with the convention that the four least-significant bits represent the fractional part while the rest represent the whole part. This approach, where we have a fixed number of digits in the fractional part, is termed a fixed-point representation. The problem however is already clear: if we have only eight bits to store our value in, there needs to be some compromise in terms of precision. That is, since we need seven bits for the whole part, there is only one bit left for the fractional part. Hence, we either truncate the number as

$$1111011.0_{(2)} = 123.0_{(10)}$$

or perform some form of rounding. This underlines the impact of our finite resources. While n limits the range of integers that can be represented, it also impacts on the precision of rational numbers since they can only ever be approximations.

Definition 29. We say that a number x is in a base-b **fixed point** representation if written as

$$x = \pm m \cdot b^c$$

where $m, c \in \mathbb{Z}$. The value m is called the **mantissa** while the fixed value c is the **exponent**; c scales m to the correct rational value and thus determines the number

of digits in the fractional part. Note that c is not included in the representation of x, it is a fixed property (or parameter) across all values.

Taking the example above, setting $b = 2$, $c = 1$ and $m = 11110110_{(2)}$ we see that

$$\begin{aligned} 11110110_{(2)} \cdot 2^{-1} &= 1111011.0_{(2)} \\ &= \quad 123.0_{(10)}. \end{aligned}$$

Not many languages have built-in types for fixed point arithmetic and there are no real standards in this area. However, despite the drawbacks, lots of applications exist where only limited precision is required, for example in storing financial or money-related values where typically only two fractional digits are required.

Floating point numbers are a more flexible approximation of rational numbers. Instead of there being a fixed number of digits in the fractional and whole parts, we let the exponent vary.

Definition 30. We say that a number x is in a base-b **floating point** representation if written as

$$x = -1^s m \cdot b^e$$

where $m, e \in \mathbb{Z}$. The value m is called the **mantissa** while e is the **exponent**; s is the **sign bit** used to determine the sign of the value.

Informally, such numbers look like

$$x = \pm \underbrace{dd \ldots dd.dd \ldots dd}_{p \text{ digits}}$$

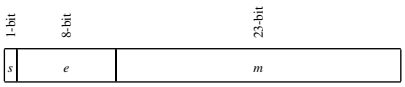
where there are p base-b digits in total. Roughly speaking, p relates to the precision of the number. We say that x is **normalised** if it is of the form

$$x = \pm \underbrace{d.dd \ldots dd}_{p \text{ digits}}$$

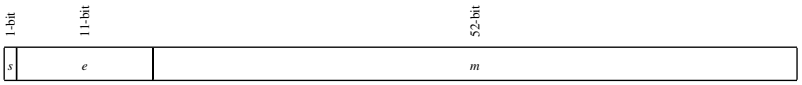
such that there is a single non-zero digit to the left of fractional point. In fact, when dealing with binary numbers if this leading digit is non-zero it *must* be one. Hence we do not need to store the leading digit and hence get one bit more of precision instead; this implicit leading digit is often called a **hidden digit**.

So that computers can inter-operate with common data formats, IEEE-754 offers two standard floating point formats which allow one to represent floating point numbers as bit-vectors. These two formats are termed single and double precision; they fit into 32-bit and 64-bit representations respectively. Both formats, described in Figure 1.3 and Listing 1.1, consist of three parts; these parts are concatenated together to form one bit-vector. A sign-magnitude method is adopted for the whole number in that a single bit denotes the sign.

The integer exponent and mantissa are both represented by bit-vectors. We need the exponent to be signed so both large and small numbers can be specified; the



(a) The 32-bit, single precision format.



(b) The 64-bit, double precision format.

Figure 1.3 Single and double precision IEEE floating point formats (as bit-fields).

```
1  struct ieee32
2  {
3    uint32 m : 23, // mantissa
4           e :  8, // exponent
5           s :  1; // sign
6  }
7
8  struct ieee64
9  {
10   uint64 m : 52, // mantissa
11          e : 11, // exponent
12          s :  1; // sign
13 }
```

Listing 1.1 Single and double precision IEEE floating point formats (as C structures).

mantissa need not be signed since we already have a sign bit for the whole number. One might imagine that a twos-complement representation of the exponent might be appropriate but the IEEE-754 standard uses an alternative approach called **biasing**. Essentially this just means that the exponent is scaled so it is always positive; for the single precision format this means it is stored with 127 added to it so the exponent 0 represents 2^{-127} while the exponent 255 represents 2^{+128}. Although this requires some extra decoding when operating with the values, roughly speaking all choices are made to make the actual arithmetic operations easier.

Again consider our example number $x = 123.3125_{(10)}$ which in binary is $x = 1111011.0101_{(2)}$. First we normalise the number to get

$$x = 1.1110110101_{(2)} \cdot 2^6$$

because now there is only one digit to the left of the binary point. Recall we do not need to store the hidden digit. So if we want a single precision, 32-bit IEEE-754 representation, our mantissa and exponent are the binary vectors

$$\begin{aligned} m &= 11101101010000000000000 \\ e &= 10000101 \end{aligned}$$

while the sign bit is 0 since we want a positive number. Using space simply to separate the three parts of the bit-vector, the representation is thus the vector

$$0 \;\; 10000101 \;\; 11101101010000000000000.$$

In any floating point representation, we need to allocate special values for certain quantities which can occur. For example, we reserve particular values to represent $+\infty$, $-\infty$ and *NaN*, or **not-a-number**. The values $+\infty$ and $-\infty$ occur when a result overflows beyond the limits of what can be represented; *NaN* occurs, for example, as a result of division by zero. For single precision, 32-bit IEEE-754 numbers these special values are

$$\begin{aligned} 0 \;\; 00000000 \;\; 00000000000000000000000 &= +0 \\ 1 \;\; 00000000 \;\; 00000000000000000000000 &= -0 \\ 0 \;\; 11111111 \;\; 00000000000000000000000 &= +\infty \\ 1 \;\; 11111111 \;\; 00000000000000000000000 &= -\infty \\ 0 \;\; 11111111 \;\; 00000100000000000000000 &= NaN \\ 1 \;\; 11111111 \;\; 00100010001001010101010 &= NaN \end{aligned}$$

with similar forms for the double precision, 64-bit format.

Finally, consider the case where the result of some arithmetic operation (or conversion) requires more digits of precision than are available. Most people learn a simple method for **rounding** in school: the value $1.24_{(10)}$ is rounded to $1.2_{(10)}$ using two digits of precision because the last digit is less than five, while $1.27_{(10)}$ is rounded to $1.3_{(10)}$ since the last digit is greater than or equal to five. Essentially, this rounds the ideal result, i.e., the result if one had infinite precision, to the most suitable representable result according to some scheme which we call the **rounding mode**. In order to compute sane results, one needs to know exactly how a given rounding mode works. IEEE-754 specification mandates the availability of four such modes; in each case, imagine we have an ideal result x. To round x using p digits of precision, one directly copies the digits $x_0 \ldots x_{p-1}$. The task of the rounding mode is then to patch the value of x_{p-1} in the right way. Throughout the following description we use decimal examples for clarity; note that essentially the same rules with minor alterations apply in binary:

Round to nearest Sometimes termed **Banker's Rounding**, this scheme alters basic rounding to provide more specific treatment when the result x lies exactly half way between two values, i.e., when $x_p = 5$ and all trailing digits from x_{p+1} onward are zero:

- If $x_p \leq 4$, then take x_{p-1} and do not alter it.
- If $x_p \geq 6$, then take x_{p-1} and alter it by adding one.
- If $x_p = 5$ and at least one of the digits x_{p+1} onward is non-zero, then take x_{p-1} and alter it by adding one.

- If $x_p = 5$ and all of the digits x_{p+1} onward are zero, then alter x_{p-1} to the nearest even digit. That is:
  - If $x_{p-1} \equiv 0 \pmod 2$, then do not alter it.
  - If $x_{p-1} \equiv 1 \pmod 2$, then alter it by adding one.

  Again considering the case of rounding for two digits of precision we find that $1.24_{(10)}$ is rounded to $1.2_{(10)}$ since the p-th digit is less than than or equal to four; $1.27_{(10)}$ is rounded to $1.3_{(10)}$ since the p-th digit is greater than or equal to six; $1.251_{(10)}$ is rounded to $1.3_{(10)}$ since the p-th digit is equal to five and the $(p+1)$-th digit is non-zero; $1.250_{(10)}$ is rounded to $1.2_{(10)}$ since the p-th digit is equal to five, digit $(p+1)$ and onward are zero, and the $(p-1)$-th digit is even; and $1.350_{(10)}$ is rounded to $1.4_{(10)}$ since the p-th digit is equal to five, digit $(p+1)$ and onward are zero, and the $(p-1)$-th digit is odd.

Round toward $+\infty$ This scheme is sometimes termed **ceiling**. In the case of a positive number, if the p-th digit is non-zero the $(p-1)$-th digit is altered by adding one. In the case of a negative number, the p-th digit onward is discarded. For example $1.24_{(10)}$ and $1.27_{(10)}$ would both round to $1.3_{(10)}$ while $1.20_{(10)}$ rounds to $1.2_{(10)}$; $-1.24_{(10)}$, $-1.27_{(10)}$ and $-1.20_{(10)}$ all round to $-1.2_{(10)}$.

Round toward $-\infty$ Sometimes termed **floor**, this scheme is the complement of the round toward $+\infty$ above. That is, in the case of a positive number the p-th digit onward is discarded. In the case of a negative number, if the p-th digit is non-zero the $(p-1)$-th digit is altered by adding one. For example, $-1.24_{(10)}$ and $-1.27_{(10)}$ would both round to $-1.3_{(10)}$ while $-1.20_{(10)}$ rounds to $-1.2_{(10)}$; $1.24_{(10)}$, $1.27_{(10)}$ and $1.20_{(10)}$ all round to $1.2_{(10)}$.

Round toward zero This scheme operates as round toward $-\infty$ for positive numbers and as round toward $+\infty$ for negative numbers. For example $1.27_{(10)}$, $1.24_{(10)}$ and $1.20_{(10)}$ round to $1.2_{(10)}$; $-1.27_{(10)}$, $-1.24_{(10)}$ and $-1.20_{(10)}$ round to $-1.2_{(10)}$.

The C standard library offers access to these features. For example the `rint` function rounds a floating point value using the currently selected IEEE-754 rounding mode; this mode can be inspected using the `fegetround` function and set using the `fesetround` function. The latter two functions use the constant values `FE_TONEAREST`, which corresponds to the round to nearest mode; `FE_UPWARD`, which corresponds to the round toward $+\infty$ mode; `FE_DOWNWARD`, which corresponds to the round toward $-\infty$ mode; and `FE_TOWARDZERO`, which corresponds to the round toward zero mode.

Using a slightly cryptic program, we can demonstrate in a very practical way that floating point representations in C work as expected. Listing 1.2 describes a C union called `ieee32` which shares space between a single precision `float` value called `f` and a 32-bit `int` called `i`; since they share the same physical space, essentially this allows us to fiddle with bits in `i` so as to provoke changes in `f`. The program demonstrates this; compiling and executing the program gives the following output:

```
bash$ gcc -std=gnu99 -o F F.c
bash$ ./F
00000000
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fenv.h>

typedef union __ieee32
{
  float  f;
  uint32 i;
}
ieee32;

int main( int argc, char* argv[] )
{
  ieee32 t;

  t.f = 2.8;

  printf( "%08X\n", ( t.i & 0x80000000 ) >> 31 );
  printf( "%08X\n", ( t.i & 0x7F800000 ) >> 23 );
  printf( "%08X\n", ( t.i & 0x007FFFFF ) >>  0 );

  t.i |= 0x80000000;
  printf( "%f\n", t.f );

  t.i &= 0x807FFFFF;
  t.i |= 0x40800000;
  printf( "%f\n", t.f );

  t.i  = 0x7F800000;
  printf( "%f\n", t.f );

  t.i  = 0x7F808000;
  printf( "%f\n", t.f );
}
```

Listing 1.2 A short C program that performs direct manipulation of IEEE floating point numbers.

```
00000080
00333333
-2.800000
-5.600000
inf
nan
```

The question is, what on earth does this mean ? The program first constructs an instance of the `ieee32` union called `t` and sets the floating point field to $2.8_{(10)}$. The `printf` statements that follow extract and print the sign, exponent and mantissa fields of `t.f` by examining the corresponding bits in `t.i`. The output shows that the sign field is $0_{(16)}$ (i.e., the number is positive), the exponent field is $80_{(16)}$ (i.e., after considering the bias, the exponent is $1_{(16)}$) and the mantissa field is $333333_{(16)}$. If we write this as a bit-vector as above, we have

$$0\ \ 10000000\ \ 01100110011001100110011$$

which is representing the actual value

$$-1^0 \cdot 1.01100110011001100110011001_{(2)} \cdot 2^1$$

or $2.8_{(10)}$ as expected. Next, the program ORs the value $80000000_{(16)}$ with `t.i` which has the effect of setting the MSB to one; since the MSB of `t.i` mirrors the sign of `t.f`, we expect this to have negated `t.f`. As expected, the next `printf` produces $-2.8_{(10)}$ to verify this. The program then ANDs `t.i` with $807FFFFF_{(16)}$ and then ORs it with $40800000_{(16)}$. The net effect is to set the exponent field of `t.f` to the value $129_{(10)}$ rather than $128_{(10)}$ as before, or more simply to multiply the value by two. Again as expected, the next `printf` produces $-5.6_{(10)} = 2 \cdot -2.8_{(10)}$ to verify this. Finally, the program sets `t.i` to the representations we would expect for $+\infty$ and *NaN* to demonstrate the output in these cases.

1.5 Toward a Digital Logic

We have already hinted that digital computers are happiest dealing with the binary values zero and one since they can be easily mapped onto low and high electrical signals. We have also seen that it is possible to define a Boolean algebra over the set $\mathbb{B} = \{0,1\}$ where the meaning of the operators $\wedge$, $\vee$ and $\neg$ is described by the following table:

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Note that we have included $x \oplus y$ to denote exclusive-or, which we call XOR. In a sense, this is just a shorthand since

$$x \oplus y = (\neg x \wedge y) \vee (x \wedge \neg y)$$

such that XOR is defined in terms of AND, OR and NOT. Now consider all the pieces of theory we have accumulated so far:

- We know that the operators AND, OR and NOT obey the axioms outlined previously and can therefore construct Boolean expressions and manipulate them while preserving their meaning.
- We can describe Boolean formula of the form

  $$\mathbb{B}^n \rightarrow \mathbb{B}$$

  and hence also construct functions of the form

  $$\mathbb{B}^n \rightarrow \mathbb{B}^m$$

simply by having m functions and grouping their outputs together. It thus makes sense that NOT, AND, OR and XOR can be well-defined for the set $\mathbb{B}^n$ as well as $\mathbb{B}$. We just define n Boolean functions such that, for example,

$$f_i : \mathbb{B} \to \mathbb{B}, \quad f_i(x,y) = x_i \wedge y_i.$$

So for $x,y,z \in \{0,1\}^n$ we can write $z_i = f_i(x,y)$ with $0 \leq i < n$ to perform the AND operation component-wise on all the bits from x and y. In fact, we do not bother with this long-winded approach and just write $z = x \wedge y$ to mean the same thing.

- We know how to represent numbers as members of the set $\mathbb{B}^n$ and have described how bit-vectors are just a short hand way of writing down such representations. Since we can build arbitrary Boolean functions of the form

$$\mathbb{B}^n \to \mathbb{B}^m,$$

the results of Shannon which show how such functions can perform arithmetic on our number representations should not be surprising. For example, imagine we want to add two integers together. That is, say $x,y,z \in \mathbb{Z}$ are represented as n-bit vectors $p,q,r \in \mathbb{B}^n$. To calculate $z = x + y$, all we need to do is make n similar Boolean functions, say f_i, such that $r_i = f_i(p,q)$. Each function takes p and q, the representation of x and y, as input and produces a single bit of r, the representation of z, as output.

In short, by investigating how Boolean algebra unifies propositional logic and set theory and how to represent numbers using sets, we have a system which is ideal for representing and manipulating such numbers using digital components. At the most basic level this is all a computer does: we have formed a crucial link between what seems like a theoretical model of computation and the first step toward a practical realisation.

1.6 Further Reading

- D. Goldberg.
  What Every Computer Scientist Should Know About Floating-Point Arithmetic.
  In *ACM Computing Surveys*, **23**(1), 5–48, 1991.
- N. Nissanke.
  Introductory Logic and Sets for Computer Science.
  Addison-Wesley, 1998. ISBN: 0-201-17957-1.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.

- K.H. Rosen.
  Discrete Mathematics and it's Applications.
  McGraw-Hill, 2006. ISBN: 0-071-24474-3.
- J. Truss.
  Discrete Mathematics for Computer Scientists.
  Addison-Wesley, 1998. ISBN: 0-201-36061-6.

1.7 Example Questions

1. For the sets $A = \{1,2,3\}$, $B = \{3,4,5\}$ and $\mathscr{U} = \{1,2,3,4,5,6,7,8\}$ compute the following:

a. $|A|$.
b. $A \cup B$.
c. $A \cap B$.
d. $A - B$.
e. $\overline{A}$.
f. $\{x : 2 \cdot x \in \mathscr{U}\}$.

2. For each of the following decimal integers, write down the 8-bit binary representation in sign-magnitude **and** twos-complement:

a. $+0$.
b. -0.
c. $+72$.
d. -34.
e. -8.
f. 240.

3. For some 32-bit integer x, explain what is meant by the Hamming weight of x; write a short C function to compute the Hamming weight of an input value.

4. Show that

$$(\neg x \wedge y) \vee (\neg y \wedge x) \vee (\neg x \wedge \neg y) = \neg x \vee \neg y.$$

Chapter 2
Basics of Digital Logic

Scientists build to learn; Engineers learn to build.

– F. Brooks

Abstract In the previous chapter we made some statements regarding various features of digital logic systems; for example, we stated that they could easily represent the binary digits zero and one because these could be mapped onto low and high electrical signals. The goal of this chapter is to expand on these statements by showing how they are satisfied using basic physics. Since most people learn about physics in school and an in-depth discussion would require another book, our goal is not rigorous accuracy but simply a rough recap and overview of the pertinent details. Using this starting point, we build higher level digital logic components capable of representing meaningful values and performing useful operations on them. We then look at how simple storage elements can be constructed and how sequences of operations can be performed using state machines and clocks.

2.1 Switches and Transistors

2.1.1 Basic Physics

Everything in the universe is composed from primitive building blocks called **atoms**. An atom in turn is composed from a group of subatomic particles: a group of nucleons, either **protons** or **neutrons**, in a central core or **nucleus** and a cloud of **electrons** which orbit the nucleus. The number of protons translates into the **atomic number** of the atom which roughly dictates what chemical element or material it is. Silicon has atomic number fourteen, for example, since there are fourteen protons in the nucleus. The arrangement of electrons around the nucleus is called the **electron configuration**; electrons can orbit the nucleus in one of several levels or **shells**.

Subatomic particles carry an associated electrical **charge** which is characteristic of their type; electrons carry a negative charge, protons have a positive charge and neutrons have neutral charge. An **ion** is an atom with a non-neutral charge overall. A negatively charged ion has more electrons in the electron cloud than protons in the nucleus, a positively charged ion has fewer electrons than protons. Electrons can be displaced from an electron cloud, using some energy, in a process called **ionisation**. How much energy is required to perform this process relates to how tightly coupled the electrons are to the nucleus and is dictated in part by the type of atom. For example, metals generally have loosely coupled electrons while plastics generally have tightly coupled electrons. We usually term these types of material **conductors** and **insulators** respectively.

Ionisation aside, electrons repel each other but are attracted by **holes** or space in an electron cloud. So given the right conditions, electrons may move around between atoms. In particular, if we apply a potential difference between two points on a conductor then electrons, and hence their associated charge, will move and create an electrical current between the points. This is a useful starting point but depends highly on the materials we are using and their electron configurations: if we do not have a material with the right number of electrons or holes, it will not behave as required. However, we can manipulate a material to have the required properties via a process called **doping**. We take the starting material and dope it, or mix it, with a second material called the **donor**. Depending on the properties of the donor, the new material will have a different number of electrons or holes but otherwise retain very similar characteristics. As an example, consider pure silicon which has an electron cloud of four electrons (only about half full): it is more or less an insulator. Doping with a boron or aluminium donor creates extra holes while doping with phosphor or arsenic creates extra electrons. We call the new materials P-type and N-type **semiconductors** respectively; the material is not a conductor or an insulator but somewhere in between.

In summary, we can more or less create materials with atomic characteristics of our choosing. The basic idea in building digital logic components from these materials is to sandwich together wafers of the materials in different ways. For example, electrons can flow from N-type to P-type semiconductors but not in the other direction; we can use this fact to build atomic scale **switches** called **transistors**.

2.1.2 Building and Packaging Transistors

Although transistors can perform a variety of functions, for example the amplification of signals, we will use them almost exclusively as switches. That is, a component that conducts, or allows charge to flow, between two terminals when we turn on the switch and not conduct, or be a resistor, when we turn off the switch. There are several different types of transistor but we will concentrate on one of the simplest, the **Field Effect Transistor (FET)** which was originally invented in 1952 by William Shockley, an engineer at Bell Labs. In this context we term the two tran-

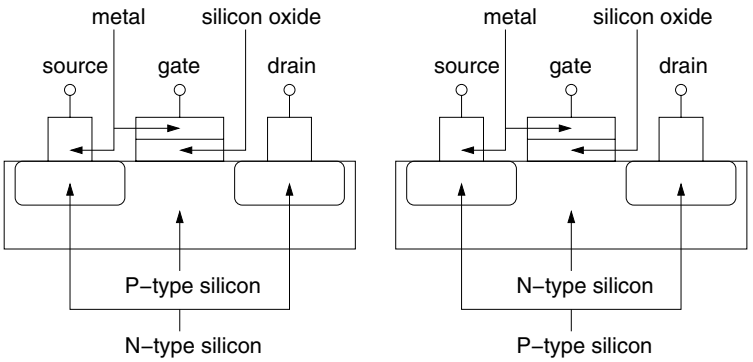


Figure 2.1 Implementation of N-MOSFET and P-MOSFET transistors.

sistor terminals the **source** and **drain**, and call the switch that controls the flow of current the **gate**.

To actually create the switch, we combine and layer materials with different properties. Again there are several ways to approach this: we concentrate on the most common design which is called a **Metal Oxide Semiconductor Field-Effect Transistor (MOSFET)**, invented in 1960 by Martin Atalla, also at Bell Labs. Figure 2.1 shows how a typical MOSFET device is built from N-type and P-type semiconductor materials and layers of metal; the two different types are termed N-MOSFET and P-MOSFET devices. The rough idea is that by applying a potential difference between the gate and source, an electric field is created which offers a channel between the source and drain through which current can flow. The type of channel depends on the types of semiconductor material used. Changing the potential difference between gate and source changes the conductivity between source and drain, and hence regulates the current: if the potential difference is small, the flow of current is small and vice versa. Thus, the gate acts like a switch that can regulate the current between source and drain.

For an N-MOSFET the types of semiconductor material used mean that when there is a high potential difference between gate and source, the conduction channel is open and current flows: the switch is on. When there is a low potential difference between gate and source, the channel closes and the switch is off. A P-MOSFET reverses this behaviour by swapping around the types of semiconductor material: when the potential difference is high the switch is off, when it is low the switch is on. Although we say the switch is off, the transistor still allows a small level of conductivity between source and drain because of the imperfect nature of the materials involved; this effect is termed **leakage**.

Transistors are seldom used in isolation and are more commonly packaged into larger components before use in larger digital circuits: these components form what are often called **logic styles**. The most frequently used style, and one of the rea-

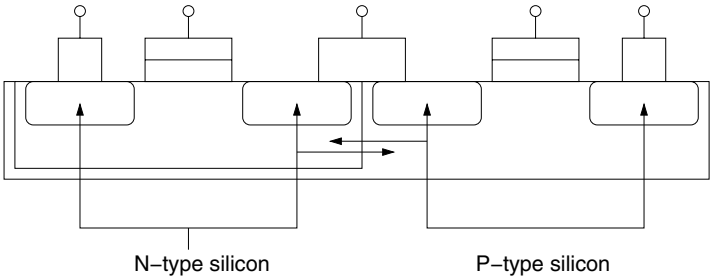
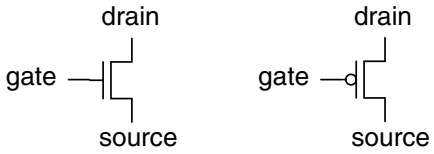


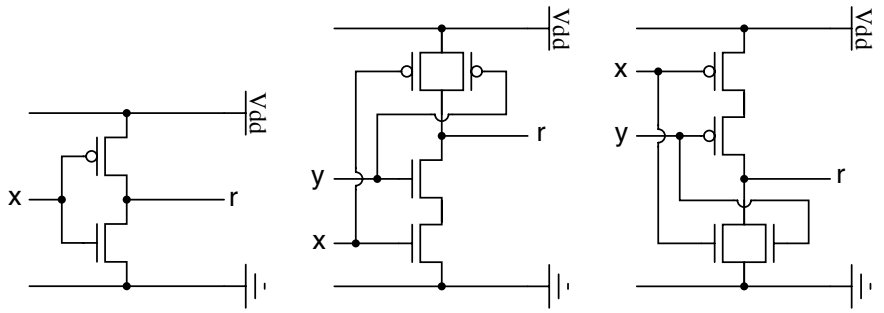
Figure 2.2 A CMOS cell built from N-MOSFET and P-MOSFET transistors.



(a) N-MOSFET. (b) P-MOSFET.

Figure 2.3 Symbolic representations of N-MOSFET and P-MOSFET transistors.

sons behind the popularity of MOSFET type transistors, is called **Complementary Metal-Oxide Semiconductor (CMOS)**; it was invented in 1963 by Frank Wanlass at Fairchild Semiconductor. The idea of CMOS is to pair each N-MOSFET transistor with a P-MOSFET partner into a component sometimes termed a **cell**. Figure 2.2 details a CMOS cell; the two transistors are arranged in a complementary manner such that whenever one is conducting, the other is not. This arrangements allows one to form circuits which have two key features: a **pull up network** of P-MOSFET transistors which sit between the positive power rail (that supplies the V_{dd} voltage level) and the output, and a **pull down network** of N-MOSFET transistors which sit between the negative power rail (which supplies the *GND* voltage level) and the output. Only one of the networks will be active at once, so aside from leakage a CMOS cell only consumes power when the inputs are **toggled** or **switched**. Since we commonly aim to build circuits with many very small transistors in close proximity to each other, this feature provides some significant advantages. In particular, it means that the use of CMOS reduces power consumption and heat dissipation which in turn aids reliability and enables size reduction. Thus, CMOS can be characterised as a low-power alternative to competitors such as **Transistor-Transistor Logic (TTL)** which is usually built from **Bipolar Junction Transistors (BJT)** and is operationally faster. It can be common to mix the characteristics of CMOS and TTL in one circuit, a technology termed bipolar-CMOS (BiCMOS).



(a) NOT. (b) NAND. (c) NOR.

Figure 2.4 Implementations of NOT, NAND and NOR logic gates using transistors.

x	y	r = NOT x	r = x NAND y	r = x NOR y
GND	GND	V_{dd}	V_{dd}	V_{dd}
GND	V_{dd}	V_{dd}	V_{dd}	GND
V_{dd}	GND	GND	V_{dd}	GND
V_{dd}	V_{dd}	GND	GND	GND

Table 2.1 Truth tables for transistor implementations of NOT, NAND and NOR.

When constructing higher-level components that utilise these behaviours, we use the symbols detailed in Figure 2.3 to represent different transistor types. We term the higher-level components we are aiming for **logic gates**, the idea being that they implement some function related to those we saw at the end of Chapter 1 and whose behaviour is described by Table 2.1.

For example, consider building a component which inverts the input; we call this a NOT gate. That is, when the input x is V_{dd} the component outputs GND and vice versa. We can achieve this using the arrangement of transistors in Figure 2.4a. To see that this works as required, consider the different states the input can be in and the properties of N-MOSFET and P-MOSFET transistors. When the input x is connected to GND the bottom transistor will be closed while the top one will be open: the output r will be connected to V_{dd}. When input x is connected to V_{dd} the bottom transistor will be open while the top one will be closed: the output will be connected to GND.

We can build two further components in a similar way; these are the NAND (or NOT AND) and NOR (or NOT OR) gates in Figure 2.4b-c. We can reason about their behaviour in a similar way as above. For example, consider the NAND gate. When input x is connected to GND the bottom transistor will be closed while the top left transistor will be open: the output r will be connected to V_{dd} no matter what input y is connected to. Likewise, when input y is connected to GND the middle

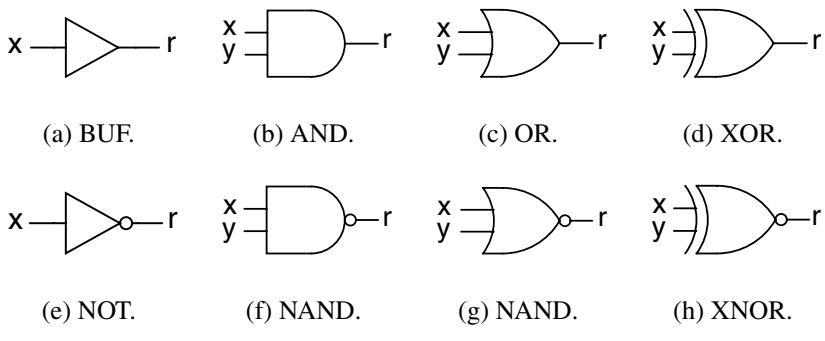


Figure 2.5 Symbolic representation of standard logic gates.

transistor will be closed while the top right transistor will be open: the output will be connected to V_{dd} no matter what input x is connected to. However, when both x and y are connected to V_{dd} the top two transistors will be closed but the bottom two will be open; this means the output is connected to *GND*. A similar reasoning applies to the operation of the NOR gate.

Clearly this is a good step toward building useful circuits in that we now have devices, constructed from very simple components, that perform meaningful operations. Specifically, we can relate the functions they implement to many of the concepts previously encountered. That is, we have a means of actually building devices to compute Boolean functions that up until now have been discussed abstractly.

2.2 Combinatorial Logic

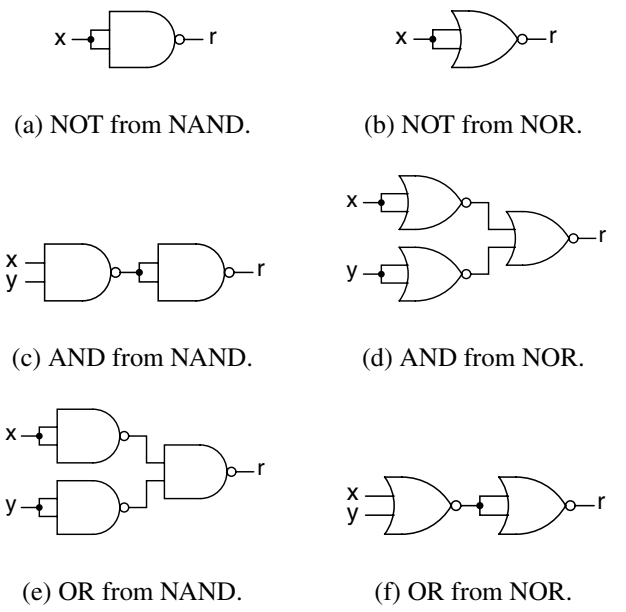
2.2.1 Basic Logic Gates

It is somewhat cumbersome to work with transistors because they represent such a low-level of detail. In order to make our life easier, we commonly adopt a more abstract view of logic gates by taking two steps: we forget about the voltage levels *GND* and V_{dd}, abstractly calling them 0 and 1, then we forget about the power rails and just draw each gate using a single symbol with inputs and outputs. The goal is that by viewing the gates more abstractly, we can focus on their properties as Boolean logic functions rather than their underlying implementation as transistors or via another technology. By composing the gates together we can build all manner of operations which are loosely termed **combinatorial logic**; the gate behaviours combine to compute the function continuously, the output is always updated as a product of the inputs.

Figure 2.5 defines symbols for each of the NOT, NAND and NOR gates built above and also the AND, OR and XOR operations described at the end of the previ-

x y	BUF $r = x$	AND $r = x \wedge y$	OR $r = x \vee y$	XOR $r = x \oplus y$
0 0	0	0	0	0
0 1	0	0	1	1
1 0	1	0	1	1
1 1	1	1	1	0
x y	NOT $r = \neg x$	NAND $r = x \mathbin{\overline{\wedge}} y = \neg(x \wedge y)$	NOR $r = x \mathbin{\overline{\vee}} y = \neg(x \vee y)$	XNOR $r = x \mathbin{\overline{\oplus}} y = \neg(x \oplus y)$
0 0	1	1	1	1
0 1	1	1	0	0
1 0	0	1	0	0
1 1	0	0	0	1

Table 2.2 Truth tables for standard logic gates.

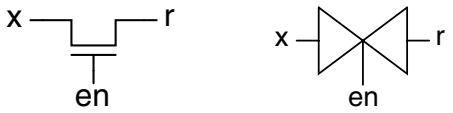


(a) NOT from NAND. (b) NOT from NOR.

(c) AND from NAND. (d) AND from NOR.

(e) OR from NAND. (f) OR from NOR.

Figure 2.6 Identities for standard logic gates in terms of NAND and NOR.



(a) Implementation. (b) Symbol.

Figure 2.7 An implementation and symbolic description of a 3-state enable gate.

ous chapter; these symbols represent functions with behaviour described as a truth table in Table 2.2. Note the use of an **inversion bubble** on the output of a gate to invert the value. Using this notation, a buffer (or BUF) is simply a gate that passes the input straight through to the output and a NOT gate just inverts the input to form the output. Also note that for completeness we have included the XNOR (or NXOR) symbol which has the obvious meaning but is seldom used in practise. We use $\overline{\wedge}$, $\overline{\vee}$ and $\overline{\oplus}$ as a short hand to denote the NAND, NOR and XNOR operations respectively. Finally, for two input gates such as AND, OR and XOR we often use a notational short hand and draw the gates with more than two inputs; this is equivalent to making a tree of two-input gates since, for example, we have that

$$(w \wedge x \wedge y \wedge z) = (w \wedge x) \wedge (y \wedge z).$$

The natural question to ask after this is why have we built NAND and NOR when AND and OR seem to be more useful ? The answer is related to the cost of implementation; it is simply more costly to build an AND gate from transistors, the cheapest way being to compose a NAND gate with a NOT gate. Since NAND and NOR are the cheapest meaningful components we can build, it makes sense to construct a circuit from these alone. This is possible because NAND and NOR are both **universal** in the sense that one can implement all the other logic gates using them alone; one simply substitutes gates using the identities in Figure 2.6 to perform the translation. The reason for wanting to use just one component to implement an entire circuit is related to how the circuit is fabricated: one can typically build much more reliable and compact circuits if they are more regular.

2.2.2 3*-state Logic*

Roughly speaking, one can freely connect the output of one logic gate to the input of another. While it does not really make sense to connect two inputs together since *neither* will drive current along the wire, connecting two outputs together is more dangerous since *both* will drive current along the wire. The outcome depends on a number of factors but is seldom good: the transistors which are wired to fight against each other typically melt and stop working.

x	en	r
0	0	Z
1	0	Z
Z	0	Z
0	1	0
1	1	1
Z	1	Z

Table 2.3 A truth table for a 3-state enable gate.

We can mitigate this fact by introducing a slightly different style of logic called 3**-state logic**; we do not go into a great deal of depth about how such logic gates are constructed but instead focus on their behaviour. The central component in 3-state logic is a switch which is sometimes called an **enable gate**. The single transistor implementation of the gate and the more common symbolic description are shown in Figure 2.7. The gate has some interesting characteristics in terms of its behaviour which is shown as a truth table in Table 2.3.

Notice that we have introduced a new logic state, hence the name 3-state, called Z or **high impedance**. When the enable signal *en* is set to 0 the gate does not pass anything through to the output r, it is not driven with anything so in a sense some other device can be connected to and drive the same wire. However, when *en* is set to 1, the gate passes through the input x to the output r: nothing else should be driving a value along this wire or we are back to the situation which caused the original melted transistor. In summary, the high impedance signal acts as a sort of "empty" value allowing any other signal to overpower it without causing damage to the surrounding transistors. Thus, careful use of enable gates allows two or more normal logic gate outputs to be connected to the same wire: as long as only one of them is enabled at a time, and hence driving a signal along the wire, there will be no disasters.

2.2.3 Designing Circuits

The next issue to address is how one combines simple Boolean logic gates to create higher-level functions. Fortunately we have already covered a lot of material which enables us to do this in a fairly mechanical form. In particular, given a truth table that describes the function required, one can perform a simple sequence of steps to extract the corresponding sum of products (SoP) expression built from a number of minterms:

1. Imagine there are n inputs numbered $1 \ldots n$, and one output:
   - Use $I_{i,j}$ to denote the value of input j in row i.

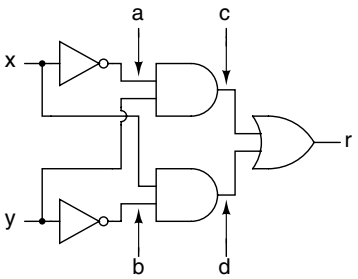


Figure 2.8 An implementation of an XOR gate.

- Use O_i to denote the value of the output in row i.

2. Find all the rows, a set R, for which given $r \in R$ we have that $O_r = 1$. That is, R is all row numbers where the output is 1.
3. For each such $r \in R$, find
   - A set H_r such that for $h \in H_r$, $I_{r,h} = 1$.
   - A set L_r such that for $l \in L_r$, $I_{r,l} = 0$.
4. The expression is then given by

$$O = \bigvee_{r \in R} \left(\bigwedge_{i \in H_r} I_{r,i} \wedge \bigwedge_{j \in L_r} \neg I_{r,j} \right).$$

The large $\wedge$ and $\vee$ symbols used here are similar to the summation and product symbols (Σ and Π). They basically AND and OR many terms together in a similar way to Σ and Π which add and multiply many terms together.

Consider the example of constructing an XOR gate whose truth table can be found in Table 2.2. Setting $x = I_1$ and $y = I_2$ while numbering the rows of the table $1 \ldots 4$, we find that we want a 1 from our expression in rows 2 and 3 and a 0 in rows 1 and 4. This is expressed using the set $R = \{2,3\}$ such that $H_2 = \{2\}$, $L_2 = \{1\}$, $H_3 = \{1\}$ and $L_3 = \{2\}$. As a result, our expression is

$$(I_2 \wedge \neg I_1) \vee (I_1 \wedge \neg I_2)$$

or rather

$$(\neg x \wedge y) \vee (x \wedge \neg y)$$

which we can verify as being correct. To build the circuit diagrammatically we simply replace each of the operators with a gate and connect the inputs and outputs together as one would expect; for our XOR expression above this is shown in Figure 2.8.

w	x	y	z	r
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Table 2.4 An example 4-input function.

2.2.4 Simplifying Circuits

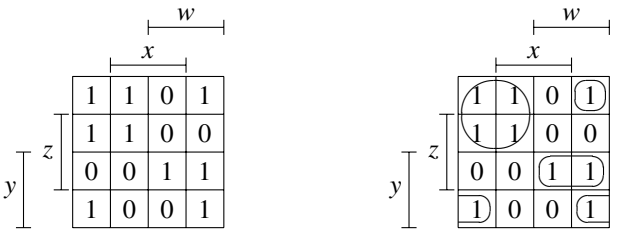
Again using the theory developed in the previous chapter, we can manipulate our XOR expression from above using the axiomatic laws of Boolean logic. For example, we can construct the product of sums (PoS) expression built from a number of maxterms

$$(x \vee y) \wedge (\neg x \vee \neg y)$$

or the functionally equivalent expression

$$(x \vee y) \wedge \neg(x \wedge y).$$

We know that if there were more than one output required, say m outputs for example, we can simply construct m such expressions using x and y which each produce one of the outputs. This raises an important issue in designing circuits: if we can simplify our expressions in the sense that they use less operators, we use less transistors and hence less space. This is a valuable step and is coupled to the issue that if we have multiple expressions, we can potentially share common blocks between the two; for example, if two expressions include some term $x \wedge y$, we only need one gate to produce the result, not two.



(a) Original Karnaugh map. (b) Annotation with minterms.

Figure 2.9 Karnaugh maps for an example 4-input function.

2.2.4.1 Karnaugh Maps

Consider the truth table in Table 2.4 describing some function r, and how one might derive a SoP expression for r to implement it. By applying the method from the previous section we get

$$\begin{aligned} r = &(\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ &(\neg w \wedge \neg x \wedge \neg y \wedge z) \vee \\ &(\neg w \wedge \neg x \wedge y \wedge \neg z) \vee \\ &(\neg w \wedge x \wedge \neg y \wedge \neg z) \vee \\ &(\neg w \wedge x \wedge \neg y \wedge z) \vee \\ &(w \wedge \neg x \wedge \neg y \wedge \neg z) \vee \\ &(w \wedge \neg x \wedge y \wedge \neg z) \vee \\ &(w \wedge x \wedge y \wedge z) \end{aligned}$$

which is verbose to say the least ! The **Karnaugh map**, a method of simplifying such expressions, was invented in 1953 by Maurice Karnaugh while working as an engineer at Bell Labs. The idea is to first provide a concise way to write down lengthy truth tables, and secondly a mechanism to easily create and simplify logic expressions from said tables without resorting to manipulation by logical axioms.

The first step is to encode the truth table on a grid as shown for our example in Figure 2.9a. The encoding used is not the natural binary ordering one might expect. For example, writing the inputs as vectors of the form (w,x,y,z) we might expect the order to follow the original truth table and produce the sequence

$$
\begin{array}{c}
(0,0,0,0)\\
(0,0,0,1)\\
(0,0,1,0)\\
(0,0,1,1)\\
(0,1,0,0)\\
(0,1,0,1)\\
(0,1,1,0)\\
(0,1,1,1)\\
\ldots
\end{array}
$$

such that moving from, for example, $(0,0,1,1)$ to $(0,1,0,0)$ flips the three bits associated with x, y and z. Instead, we use what is called a **Gray code**, named after Frank Gray who patented the idea in 1953 while also working as a researcher at Bell Labs; such codes had been known and used for a couple of hundred years before that. The basic idea is that between neighbouring lines in the sequence only one bit should change; thus we get a sequence such as

$$
\begin{array}{c}
(0,0,0,0)\\
(0,0,0,1)\\
(0,0,1,1)\\
(0,0,1,0)\\
(0,1,1,0)\\
(0,1,1,1)\\
(0,1,0,1)\\
(0,1,0,0)\\
\ldots
\end{array}
$$

such that moving from $(0,1,1,1)$ to $(0,1,0,1)$ flips only the bit associated with y. In fact, moving from any line to the previous or next flips only one bit. Looking at the Karnaugh map, the left most column represents the values where $(w,x) = (0,0)$ and reading from top to bottom where $(y,z) = (0,0)$, $(0,1)$, $(1,1)$, $(1,0)$. The next columns are similar for y and z but for the cases where $(w,x) = (0,1)$, $(1,1)$ and $(1,0)$.

Use of this technique to encode the Karnaugh map grid allows us to easily simplify the function in question. The basic idea is that by locating minterms which differ in only one input, we can simplify the resulting expression by eliminating this input. In our example, the minterms $(1,0,1,0)$ and $(1,0,1,1)$ offer a good demonstration of this fact. We could implement these minterms using the expressions

$$w \wedge \neg x \wedge y \wedge \neg z$$

and

$$w \wedge \neg x \wedge y \wedge z.$$

This is clearly wasteful however; it does not matter what value z takes since the overall function is still 1 as long as $w = 1$, $x = 0$ and $y = 1$. Thus we can eliminate

x	y	z	r
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	?
1	1	0	1
1	1	1	?

Table 2.5 An example 3-input function.

the z term from our expressions above and simply use the single and more simple minterm

$$w \wedge \neg x \wedge y$$

to cover both cases.

The next step in realising this simplification from our Karnaugh map is to gather cells in the grid whose entry is 1 into rectangular groups of size 2^n for some integer n. We can thus form groups of 1 cell, 2 cells, 4 cells and so on: the key thing is that the groups are rectangular. Due to the properties of the Gray code, these groups are allowed to wrap around the sides of the grid; this is valid since moving right from $(w,x,y,z) = (0,0,0,0)$ in the top left-hand corner to $(1,0,0,0)$ in the top right-hand corner still only flips one bit. Additionally, groups are allowed to overlap if they want as long as they are still rectangular. The annotated Karnaugh map in Figure 2.9b demonstrates these concepts by forming four groups.

Each of the final groups represents one term we need to implement in order to implement the overall function. The bigger the group, the less variables we need to include in each of the terms. Consider, for example, the group of four cells in the top left-hand corner. Looking at how the variables are assigned values in the grid, it does not matter what value x takes as long as $w = 0$ and it does not matter what value z takes as long as $y = 0$ because we still get a 1 as output. We can implement this term as $\neg w \wedge \neg y$ to cover all four cells in that group; this is much simpler than the four separate corresponding terms in our original implementation. Applying the same technique to the function as a whole, we find that

$$\begin{aligned} r = {} & (\ \neg w \wedge \neg y \ ) \vee \\ & (\ \ w \wedge \neg x \wedge \neg y \wedge \neg z\ ) \vee \\ & (\ \ \neg x \wedge \ \ y \wedge \neg z\ ) \vee \\ & (\ \ w \wedge \ \ y \wedge \ \ z\ )\ . \end{aligned}$$

Clearly this is significantly simpler than our initial effort and does not require tedious application of logical axioms. However, it is not the most simple description of the function; we leave this issue open to explore in the next section.

(a) Grouped without don't care state. (b) Grouped with don't care state.

Figure 2.10 Karnaugh maps for an example 3-input function.

Karnaugh maps can represent functions with any number of inputs but drawing them for more than eight or so becomes tricky. More importantly, there is no reason why they need to be square. For the three input function *r* in Table 2.5 we can draw the grid in Figure 2.10a. Another important feature is detailed in this example in that some of the outputs are neither 0 nor 1 but rather ?. We call ? the **don't care state** in the sense it can take any value we want; it is not that we do not know the value, we just do not care what the value is. One can rationalise this by thinking of a circuit whose output just does not matter given some combination of input, maybe this input is invalid and so the result is never used. Either way, we can use these don't care states to our advantage during simplification. By regarding them as 1 values when we want to include them in our groupings and 0 otherwise, we can further simplify our implementation. This can be seen in Figure 2.10b where without the ? state in the middle we are forced to implement two groups but by including it we can group four cells together into one: remember, less groups and larger groups will typically mean a simpler implementation.

2.2.4.2 Quine-McCluskey

Although the Karnaugh map method for simplification is attractive, it falls down when either the number of inputs grows too large or it needs to be implemented in software. In these cases we prefer **Quine-McCluskey minimisation**, a method developed independently by Willard Quine [55] and Edward McCluskey [39] in the mid 1950s. As an example of applying the method, we re-simplify the function *r* used above and detailed in Table 2.4.

The first step is to extract all the minterms from the truth table, i.e., all the combinations of input that result in a 1 in the output. We assign a number to each minterm and write them, using vectors of the form (w,x,y,z) in this case, within a table; this is shown in the top, first phase section of Table 2.6. Placed in this form, each entry is called an **implicant**. We place the implicants into groups according to the number of 1 values in their vector representation. For example, implicant 0 corresponding to $(0,0,0,0)$ has no 1 values and is placed in group 0; implicants 1, 2, 4 and 8 all have one 1 value and are all placed in group 1.

Phase 1

0	$(0,0,0,0)$	$\surd$
1	$(0,0,0,1)$	$\surd$
2	$(0,0,1,0)$	$\surd$
4	$(0,1,0,0)$	$\surd$
8	$(1,0,0,0)$	$\surd$
5	$(0,1,0,1)$	$\surd$
10	$(1,0,1,0)$	$\surd$
11	$(1,0,1,1)$	$\surd$
15	$(1,1,1,1)$	$\surd$

Phase 2

$0+1$	$(0,0,0,-)$	$\surd$
$0+2$	$(0,0,-,0)$	$\surd$
$0+4$	$(0,-,0,0)$	$\surd$
$0+8$	$(-,0,0,0)$	$\surd$
$1+5$	$(0,-,0,1)$	$\surd$
$4+5$	$(0,1,0,-)$	$\surd$
$2+10$	$(-,0,1,0)$	$\surd$
$8+10$	$(1,0,-,0)$	$\surd$
$10+11$	$(1,0,1,-)$	
$11+15$	$(1,-,1,1)$	

Phase 3

$0+1+4+5$	$(0,-,0,-)$	
$0+2+8+10$	$(-,0,-,0)$	
$0+4+1+5$	$(0,-,0,-)$	duplicate
$0+8+2+10$	$(-,0,-,0)$	duplicate

Table 2.6 Step one of a Quine-McCluskey-based simplification: extraction of prime implicants.

Recall from our simplification using Karnaugh maps that we were able to apply a rule to cover the pair of minterms

$$w \wedge \neg x \wedge y \wedge \neg z$$

and

$$w \wedge \neg x \wedge y \wedge z$$

with one simpler minterm

$$w \wedge \neg x \wedge y$$

because the value of z has no influence on the result. We use a similar basic approach here. We compare each member of the i-th group with each member of the $(i+1)$-th group and look for pairs which differ in only one place; there is no point in comparing members of the i-th group with other groups since we know they cannot

	0	1	2	4	8	5	10	11	15
$0+1+4+5$	√	√		√		√			
$0+2+8+10$	√		√		√		√		
$10+11$							√	√	
$11+15$								√	√

Table 2.7 Step two of a Quine-McCluskey-based simplification: the prime implicants table.

satisfy this criterion since they have different numbers of 1 values. So for example, we compare implicant 0 from group 0 with implicants 1, 2, 4 and 8 from group 1 and find that in each case the pairs differ in just one place. We write these pairs in a new table; this is detailed in the middle, second phase section of Table 2.6. In the new table, we replace the differing value with a $-$ so that, for example, combining implicants 0 and 1, represented by $(0,0,0,0)$ and $(0,0,0,1)$, produces implicant $0+1$ which is represented by $(0,0,0,-)$. Furthermore, when we have used an implicant from the original table to produce a combined one in the new table we place a tick next to it; all of implicants 0, 1, 2, 4 and 8 are ticked as a result of our initial comparisons between groups 0 and 1. This process is repeated for other groups, so groups 1 and 2, 2 and 3 and 3 and 4 in this case. We then iterate the whole procedure to construct further tables until we can no longer combine any implicants. In this case we terminate after three phases as shown in Table 2.6. However, notice that in the third phase we start to introduce duplicate implicants; these duplicates are deleted from further consideration upon detection.

Implicants in any table which are unticked, i.e., have not been combined and used to produce further implicants, are called **prime implicants** and are the ones we need to consider when constructing the final expression for our function. In our example, we have four prime implicants

$$\begin{array}{ll} 0+1+4+5 & (0,-,0,-) \\ 0+2+8+10 & (-,0,-,0) \\ 10+11 & (1,0,1,-) \\ 11+15 & (1,-,1,1) \end{array}$$

To further simplify the resulting expression, the Quine-McCluskey method uses a second step. A so-called prime implicant table is constructed which lists the prime implicants along the side and the original minterms along the top. In each cell where the prime implicant includes the corresponding minterm, we place a $\surd$; this is shown in Table 2.7 for our example. The goal now is to select a combination of the prime implicants which includes, or covers, all of the original minterms. For example, the implicant $0+1+4+5$ covers the prime implicants 0, 1, 4 and 5 so selecting this as well as implicant $10+11$ will cover 0, 1, 4, 5, 10 and 11 in total.

Before we start however, we can identify so-called **essential prime implicants** as being those which are the only cover for a given minterm. Minterm 15 in our example demonstrates this case since it is only covered by prime implicant $11+15$;

it is essential to include this implicant in our final expression as a result. Starting with the essential prime implicants, we can draw a line through the associated row in the prime implicants table. Each time the line goes through a $\surd$, we also draw a line through that column. The intuition behind this is that the resulting lines show which minterms are currently covered by prime implicants we have selected for inclusion in our final expression. Following this procedure in our example, we find that using only implicants $11+15$, $0+1+4+5$ and $0+2+8+10$ we can cover all the original minterms. Looking at the associated vectors, we have

$$\begin{array}{ll} 0+1+4+5 & (0,-,0,-) \\ 0+2+8+10 & (-,0,-,0) \\ 11+15 & (1,-,1,1). \end{array}$$

Treating the entries with $-$ as don't care, we thus implement the final expression for the function r as

$$\begin{array}{rl} r = & (\ \neg w \quad \wedge \neg y \quad\ \ ) \vee \\ & (\quad \neg x \wedge \quad\ \ \neg z\ ) \vee \\ & (\ \ w \quad \wedge \ \ y \wedge \ \ z\ ) \end{array}$$

which is simpler than our original attempt using Karnaugh maps; we only have three terms in this case ! The reason is obvious when one looks at the original Karnaugh map: we formed a group of 1 cell in the top right-hand corner and a group of 2 cells from the bottom left and right-hand corners. We can easily merge these into a single group of 4 cells which covers all 4 corner cells of the map and overlaps with the group of 4 cells already in the top left-hand corner. Since less groups and larger groups mean simpler implementation, we get a simpler expression: the two terms

$$w \wedge \neg x \wedge \neg y \wedge \neg z$$

and

$$\neg x \wedge y \wedge \neg z$$

have been merged into the single term

$$\neg x \wedge \neg z$$

which covers both.

Hopefully it is clear that the Quine-McCluskey method offers something that Karnaugh maps do not. Although Karnaugh maps are a useful tool for simplifying functions with a small number of inputs by hand, Quine-McCluskey is easy to automate and hence ideal for implementation on a computer which can deal with many more inputs as a result. However, the complexity of the algorithm dictates that the upper limit in terms of number of inputs is quickly reached; logic simplification is classed as an NP-complete problem, and the complexity of Quine-McCluskey is exponential in the number of inputs.

2.2.5 Physical Circuit Properties

2.2.5.1 Propagation Delay

CMOS-based logic gates do not operate instantaneously because of the way transistor switching works at the physical level; there is a small delay between when the inputs are supplied and when the output is produced called **propagation delay**. The problem of propagation delay in gates, so-called **gate delay**, is compounded by the fact that there is also a delay associated with the wires that connect them together. A **wire delay** is typically much smaller than gate delay but the same problem applies: a value takes an amount of time proportional to the wire length to travel from one end to the other. Although such delays are extremely small, when many gates are placed in series or wires are very long the delays can add up and produce the problem of circuit **metastability**.

Consider for example the XOR circuit developed previously. So far we have taken a static view of the circuit in the sense that we set some inputs and by assuming that all gates operate instantly, compute the output. The picture changes when we take a dynamic view that includes time; in particular, we see that depending on which time period we sample, the circuit can be in a metastable state which does not correctly represent the required final result. Given the labelled XOR implementation in Figure 2.8, Figure 2.11 highlights this effect. We start at time $t = 70$ns when the circuit is in the correct state given inputs of $x = 0$ and $y = 1$. The inputs are then toggled to $x = 1$ and $y = 1$ at $t = 80$ns but the result does not appear instantly given example propagation delays for NOT, AND and OR gates as 10ns, 20ns and 20ns respectively. In particular, we can examine points in the time-line and show that the final and intermediate results are wrong. For example, it takes until $t = 90$ns before the NOT gates produce the correct outputs and the final result does not change to the correct value until $t = 130$ns; before then the output is wrong in the sense that it does not match the inputs.

One can try to equalise the delay through different paths in the circuit using dummy or **buffer** gates. A buffer simply copies the input to the output and can also amplify the input in some cases; essentially it is performing the identity function. Since the buffer takes some time to operate just like any other gate yet performs no change in input, one can place them throughout the circuit in order to ensure values arrive at the same time. An example is shown in Figure 2.12. Although the circuit may still enter metastable states, we have tamed the problem to some extent by ensuring that values passing into the two AND gates arrive at the same time.

However, even when the effects of propagation delay are minimised it produces a secondary feature that acts as a limit. This feature is the **critical path** of the circuit, the longest sequential sequence of gates in the circuit that dominates the time taken for it to produce a result. In the XOR circuit above, the critical path goes through a NOT gate, an AND gate and then an OR gate: the path has a total delay of 50ns. We will see later how this feature limits the performance of our designs. For now it is enough to believe that the shorter the critical path is, the quicker we can compute

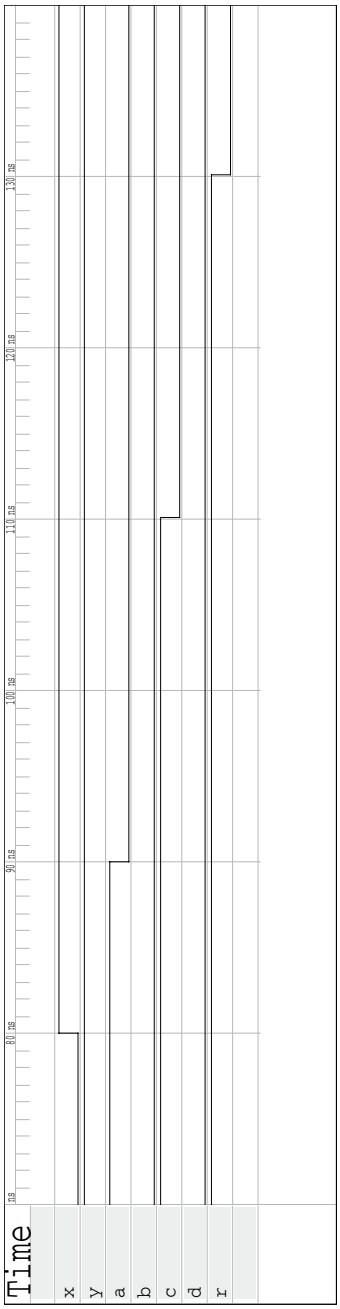


Figure 2.11 A behavioural time-line demonstrating the effects of propagation delay on the XOR implementation.

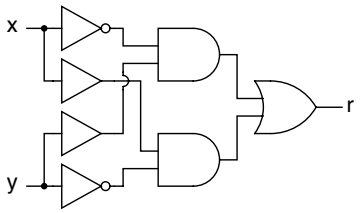


Figure 2.12 Buffering inputs within the XOR gate implementation to equalise propagation delay.

results with our circuit; hence it is advantageous to construct or simplify a design so as to minimise the critical path length.

2.2.5.2 Fan-in and Fan-out

Fan-in and **fan-out** are properties of logic gates relating to the number of inputs and outputs. The problem associated with fan-in and fan-out is that a given logic gate will have physical constraints on the amount of current which can be driven through it. For example, it is common for the output of a source gate to be connected to inputs of several target gates. Unless the source gate can drive enough current onto the output, errors will occur because the target gates will not receive enough of a share to operate correctly; fan-out is roughly the number of target gates which can be connected to a single source. CMOS-based gates have quite a high fan-out rating, perhaps 100 target gates or more can be connected to a single source.

2.2.6 Basic Building Blocks

2.2.6.1 Half and Full Adders

Ultimately, our goal in constructing digital circuits is to do some computation or, more specifically, arithmetic on numbers. The most basic key component one can think of in this context is a 1-bit adder which takes two 1-bit values, say x and y, and adds them together to product a sum and a carry, say s and co. This is commonly termed a **half-adder**, the truth table is shown in Table 2.8a. The truth table makes intuitive sense: if we add $x = 0$ to $y = 1$, the sum is 1 and there is no carry-out; if we add $x = 1$ to $y = 1$, the result is 2 which we cannot represent in 1-bit so the sum is 0 and the carry-out is 1. The half-adder is easily implementable as shown by Figure 2.13a. Essentially we use the same techniques as developed above to extract an expression for each of the outputs in terms of x and y:

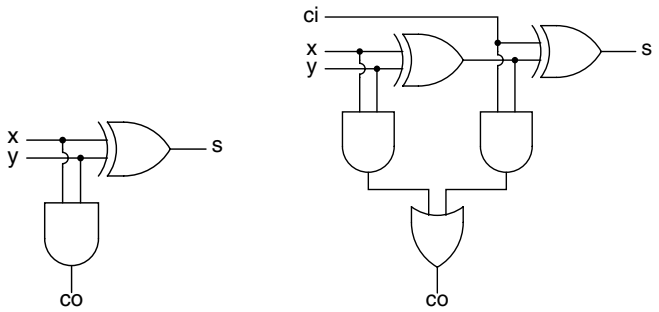
x	y	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(a) Half-adder

ci	x	y	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b) Full-adder

Table 2.8 Truth tables for 1-bit adders.



(a) Half-adder (b) Full-adder

Figure 2.13 Implementation of a 1-bit half-adder and full-adder.

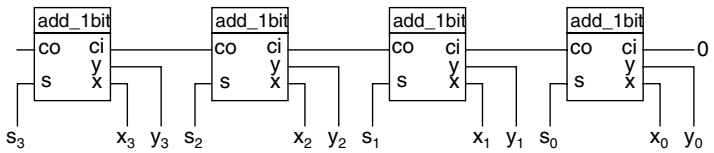
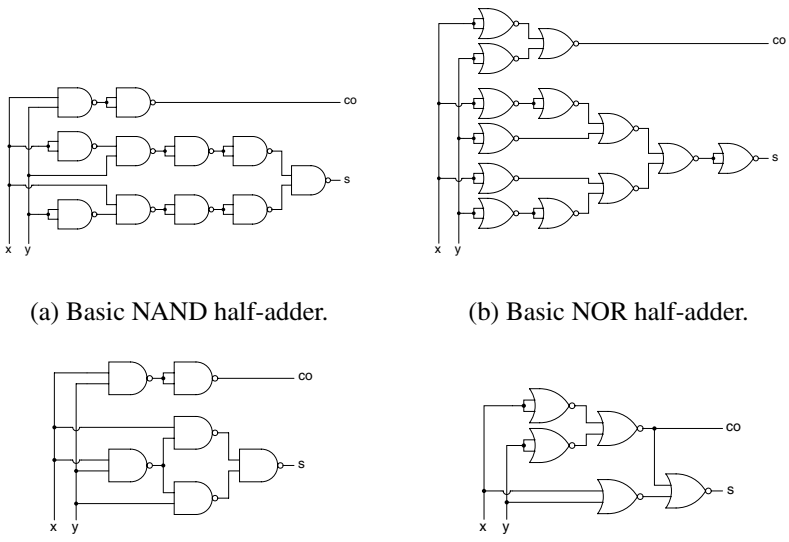


Figure 2.14 A chain of full-adders used to add 4-bit values.



(a) Basic NAND half-adder.

(b) Basic NOR half-adder.

(c) Simplified NAND half-adder.

(d) Simplified NOR half-adder.

Figure 2.15 NAND and NOR gate implementations of a half-adder.

$$\begin{aligned} co &= x \wedge y \\ s &= x \oplus y. \end{aligned}$$

However, the half-adder is only partly useful since in order to add together numbers larger than 1-bit we need a **full-adder** circuit that can accept a carry-in as well as produce a carry-out. The truth table for a full-adder is shown in Table 2.8b. Again extracting an expression for each of the outputs, this time in terms of x, y and ci we find

$$\begin{aligned} co &= (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s &= x \oplus y \oplus ci \end{aligned}$$

which are a little more complicated. To simplify these expressions we note that there is a shared term $x \oplus y$ which we only need one set of logic for. This produces the circuit shown in Figure 2.13b. Note that the full-adder is essentially two half-adders joined together. Using such components, we will see later that one can build a chain of n 1-bit full-adders that is capable of adding together n-bit values. The basic structure follows that in Figure 2.14 where each of the i full-adders takes bits i of the two inputs x and y and produces bit i of the result; the carry-out of the i-th full-adder is fed into the carry-in of $(i+1)$-th full-adder.

Finally, as an example of how to implement a circuit using only NAND or NOR gates consider translating the half-adder implementation above. Essentially we just use the identities discussed previously in place of each gate in the circuit; this is shown in Figure 2.15a-b. However, once this translation is completed we can start

x	y	equ
0	0	1
0	1	0
1	0	0
1	1	1

(a) Equality.

x	y	lth
0	0	0
0	1	1
1	0	0
1	1	0

(b) less than.

Table 2.9 Truth tables for 1-bit comparators.

to simplify the equations, eliminating and sharing logic. The results are shown in Figure 2.15c-d. We can re-write the expressions for the NOR version of the half-adder as follows:

$$\begin{aligned} co &= \neg x \mathbin{\overline{\vee}} \neg y \\ s &= \neg((\neg\neg x \mathbin{\overline{\vee}} \neg y) \mathbin{\overline{\vee}} (\neg x \mathbin{\overline{\vee}} \neg\neg y)) \\ &= \neg((x \mathbin{\overline{\vee}} \neg y) \mathbin{\overline{\vee}} (\neg x \mathbin{\overline{\vee}} y)) \\ &= (x \mathbin{\overline{\vee}} y) \mathbin{\overline{\vee}} (\neg x \mathbin{\overline{\vee}} \neg y) \end{aligned}$$

from which we can then share the term $\neg x \mathbin{\overline{\vee}} \neg y$ between co and s. This reduces the number of NOR gates used from thirteen in the original, naive translation to five in the simplified version.

2.2.6.2 Comparators

In the same way as one might want to perform arithmetic on numbers, comparison of numbers is also a fundamental building block within many circuits. With our adder circuits, we looked at the idea of addition of 1-bit numbers with a view to adding n-bit numbers later on. We take the same approach to comparison of numbers by looking at some very simple circuits for performing 1-bit equality and less than comparisons; it turns out we can produce all other comparisons from these two.

Table 2.9 shows the truth tables for 1-bit **equality** and **less than** comparison between two numbers x and y. Both operations produce a 1-bit true or false result, named equ and lth respectively. The tables are self explanatory although maybe a little odd given we are dealing with 1-bit numbers: for example, reading the table for less than we have that 0 is not less than 0, 0 is less than 1, 1 is not less than 0 and 1 is not less than 1. Using the truth tables we can easily produce the following expressions for the two operations:

$$\begin{aligned} equ &= \neg(x \oplus y) \\ lth &= \neg x \wedge y. \end{aligned}$$

s_0	i_1	i_0	r
0	?	0	0
0	?	1	1
1	0	?	0
1	1	?	1

(a) 2-way, 1-bit multiplexer.

s_1	s_0	i_3	i_2	i_1	i_0	r
0	0	?	?	?	0	0
0	0	?	?	?	1	1
0	1	?	?	0	?	0
0	1	?	?	1	?	1
1	0	?	0	?	?	0
1	0	?	1	?	?	1
1	1	0	?	?	?	0
1	1	1	?	?	?	1

(b) 4-way, 1-bit multiplexer.

Table 2.10 Truth tables for 2-way and 4-way, 1-bit multiplexers.

2.2.6.3 Multiplexers and Demultiplexers

A **multiplexer** is a component that takes n control inputs and uses them to decide which one of 2^n inputs is fed through to the single output. A **demultiplexer** does the opposite, it still takes n control inputs but uses them to decide which one of 2^n outputs is fed with the single input. We say that a multiplexer or demultiplexer is n-way if it accepts or produces n inputs or outputs, and m-bit if those inputs and outputs are m bits in size.

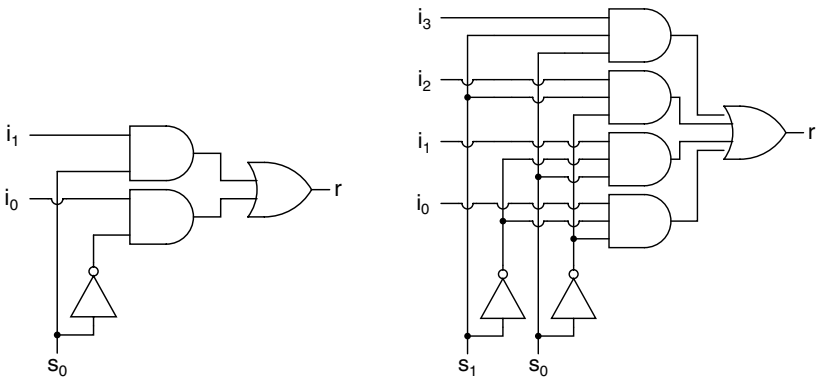
Table 2.10a shows the truth table for a 2-way, 1-bit multiplexer with inputs i_0 and i_1, a control signal s_0 and an output r. The table shows that when $s_0 = 0$ we feed the value of i_0 through to the output, i.e., $r = i_0$, while if $s_0 = 1$ we have that $r = i_1$. This behaviour is easily implemented using the expression

$$r = (\ \neg s_0 \wedge i_0\ ) \vee (\ s_0 \wedge i_1\ )$$

which is shown diagrammatically in Figure 2.16a.

A natural question arises given this simple starting point, namely how do we extend the design to produce n-way, m-bit alternatives ? Building an m-bit multiplexer is easy, we simply have m separate 1-bit multiplexers where multiplexer i takes the i-th bit of each input and produces the i-th bit of the output. This technique is the same no matter how many inputs we have and is sometimes termed **bit-slicing** since we chop up the work into bit-sized chunks.

We can produce a multiplexer design with double the number of inputs using two techniques: either produce a large truth table along the same lines as Table 2.10a but with four inputs and two control signals, or re-use our 2-way multiplexer. Considering the truth table for a 4-way multiplexer in Table 2.10b, we can derive an expression for r as



(a) 2-way, 1-bit multiplexer. (b) 4-way, 1-bit multiplexer.

Figure 2.16 Implementations for 2-way and 4-way, 1-bit multiplexers.

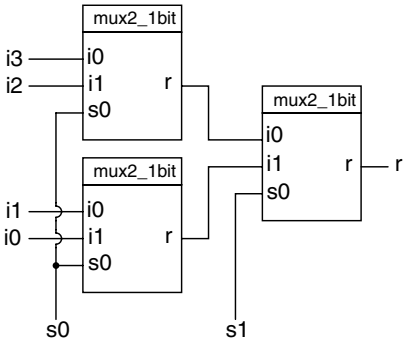


Figure 2.17 Using cascading to build a 4-way multiplexer from 2-way building blocks.

$$
\begin{aligned}
r = &(\neg s_0 \wedge \neg s_1 \wedge i_0) \vee \\
&(s_0 \wedge \neg s_1 \wedge i_1) \vee \\
&(\neg s_0 \wedge s_1 \wedge i_2) \vee \\
&(s_0 \wedge s_1 \wedge i_3) \ .
\end{aligned}
$$

This is more complicated than our 2-way multiplexer but still offers easy and efficient implementation as shown in Figure 2.16b; hopefully it is clear that one can take this general pattern and extend it to even more inputs. However, in doing so the truth table and corresponding implementation become more and more complicated to the point of being unmanageable. To address this issue we can simply re-use multiplexers with smaller numbers of inputs to build components with more inputs using a technique called **cascading**. The idea is to split the large decisional task of

s_0	i	r_1	r_0
0	0	?	0
0	1	?	1
1	0	0	?
1	1	1	?

(a) 2-way, 1-bit demultiplexer.

s_1	s_0	i	r_3	r_2	r_1	r_0
0	0	0	?	?	?	0
0	0	1	?	?	?	1
0	1	0	?	?	0	?
0	1	1	?	?	1	?
1	0	0	?	0	?	?
1	0	1	?	1	?	?
1	1	0	0	?	?	?
1	1	1	1	?	?	?

(b) 4-way, 1-bit demultiplexer.

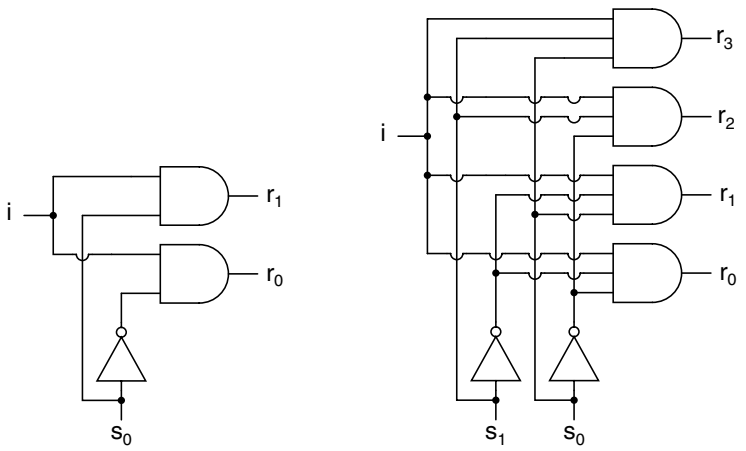
Table 2.11 Truth tables for 2-way and 4-way, 1-bit demultiplexers.

which input to feed through into smaller tasks. Figure 2.17 demonstrates this idea by building a 4-way multiplexer from 2-way multiplexer building blocks. The control signal s_0 is now used to manage the first two multiplexers; the top one produces i_3 if $s_0 = 1$ or i_2 or $s_0 = 0$, the bottom one produces i_1 if $s_0 = 1$ or i_0 or $s_0 = 0$. These outputs are fed into a final multiplexer which uses s_1 to select the appropriate input. For example, if $s_0 = 0$ and $s_1 = 1$ the top multiplexer produces i_2 while the bottom one produces i_0; the final multiplexer selects i_2 and feeds it to the output; this is what we expect for the given control signals. By using a similar approach one can construct multiplexers with even more inputs, for example a 16-way component using 4-way building blocks.

Having discussed multiplexers, demultiplexers are somewhat trivial since they are essentially just multiplexers in reverse; Table 2.11 details the truth tables for the 2-way and 4-way, 1-bit components which are implemented in Figure 2.18. Note that the same techniques of bit-slicing and cascading can be applied to demultiplexers as to multiplexers in order to produce n-way, m-bit components.

2.2.6.4 Encoders and Decoders

What we normally called **encoder** and **decoder** devices are, roughly speaking, like specialisations of the multiplexer and demultiplexer devices considered earlier. Strictly speaking we usually say decoders translate, or map, an n-bit input into one of 2^n possible output values; encoders perform the converse by translating one of 2^n possible input values into an n-bit output. Of course, some inputs or outputs might be ignored in order to relax this restriction; more generally we say an encoder or decoder is an n-**to-**m device if it has n inputs and m outputs, a 4-to-2 encoder for example. Some decoders (resp. encoders) demand that exactly one bit of their output (resp. input) is 1 at a time, these are normally called **one-of-many** devices although the term is often dropped when the assumption is that all devices are one-of-many.



(a) 2-way, 1-bit demultiplexer. (b) 4-way, 1-bit demultiplexer.

Figure 2.18 Implementations for 2-way and 4-way, 1-bit demultiplexers.

i_1	i_0	r_3	r_2	r_1	r_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

(a) 2-input one-of-many decoder.

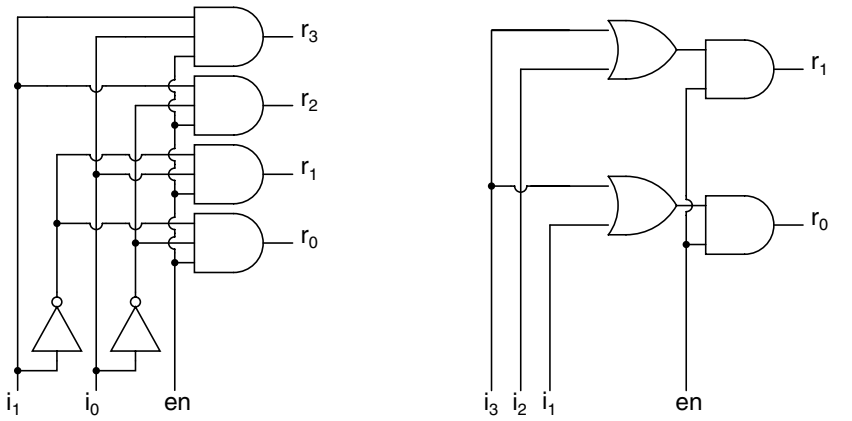
i_3	i_2	i_1	i_0	r_1	r_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(b) 2-output one-of-many encoder.

Table 2.12 Truth tables for a simple 2-input decoder and 2-output encoder.

Table 2.12a details the truth table for a simple one-of-many decoder circuit; notice that the structure is similar to a demultiplexer with the inputs tied to suitable values. This type of decoder is often used to control other circuits. Consider the case of having say four circuits in a system, only one of which should be active at a time depending on some control value. A decoder could implement this behaviour by taking the control value as input and driving 1 onto an output to activate the associated circuit. The decoder drives 1 onto the output number associated with the inputs i_0 and i_1. That is, if one considers the 2-bit value $i = (i_1, i_0)$, the decoder drives 1 onto output number i and 0 onto all other outputs.

Considering this example again, there may be occasions where no circuit should be active; it is therefore typical to equip the decoder with an **enable** input which, when set to 0, forces all outputs to 0 thus disabling any control behaviour. This is easily realised by adding an extra input *en* to the circuit which is ANDed with the outputs. Expressions for $r_0 \ldots r_3$ can be written as



(a) 2-input one-of-many decoder. (b) 2-output one-of-many encoder.

Figure 2.19 Implementation of a simple 2-input decoder and 2-output encoder.

$$\begin{aligned} r_0 &= en \wedge \neg i_0 \wedge \neg i_1 \\ r_1 &= en \wedge \quad i_0 \wedge \neg i_1 \\ r_2 &= en \wedge \neg i_0 \wedge \quad i_1 \\ r_3 &= en \wedge \quad i_0 \wedge \quad i_1 \end{aligned}$$

with the resulting implementation shown in Figure 2.19a.

Table 2.12a details the truth table for the corresponding encoder circuit which is implemented in Figure 2.19a. The idea is to perform the converse translation: take the inputs $i = (i_3, i_2, i_1, i_0)$ and decide which value the decoder would have mapped them to. This is somewhat easier to construct because the many-of-one property ensures exactly one of $i_0 \ldots i_3$ is equal to 1; expressions for r_0 and r_1 can be written as

$$\begin{aligned} r_0 &= en \wedge (i_1 \vee i_3) \\ r_1 &= en \wedge (i_2 \vee i_3) \end{aligned}$$

with the resulting implementation shown in Figure 2.19b.

In this context, it is also useful to consider what is termed a **priority encoder**. In a one-of-many encoder, exactly one input bit is allowed to be 1 but in practise this is hard to ensure. To cope with this fact, we might like to give some priority to the various possibilities. For example, if both bits $i_0 = 1$ and $i_1 = 1$ then the normal one-of-many encoder fails, we might like to say for example that we give priority to i_1 in this case and set the output accordingly. More generally, we are saying that input bit i_{j+1} has a higher priority that input bit i_j (although other schemes are obviously possible). Reading the truth table detailed in Table 2.13 enforces this relationship.

i_3	i_2	i_1	i_0	r_1	r_0
0	0	0	?	0	0
0	0	1	?	0	1
0	1	?	?	1	0
1	?	?	?	1	1

Table 2.13 Truth table for a simple 2-output priority encoder.

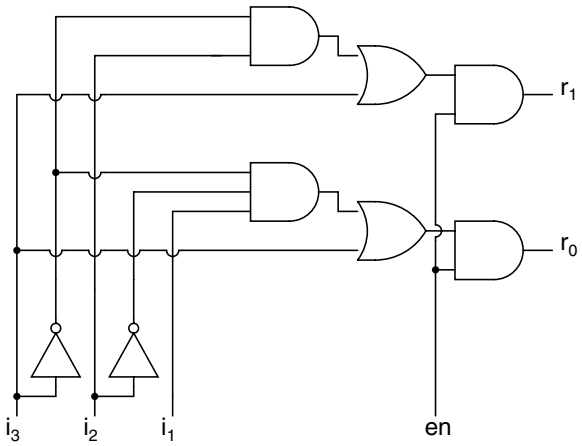


Figure 2.20 Implementation of 2-output priority encoder.

Take the bottom row for example: although potentially $i_0 = 1$, $i_1 = 1$ or $i_2 = 1$ the output gives priority to i_3. The resulting implementation of the expressions

$$\begin{aligned} r_0 &= (i_1 \wedge \neg i_2 \wedge \neg i_3) \vee (i_3) \\ r_1 &= (i_2 \wedge \neg i_3) \qquad\quad \vee (i_3) \end{aligned}$$

is shown in Figure 2.20.

As a concrete example, consider the task of controlling a 7-segment LED as found in many calculators and shown in Figure 2.21. The device has seven segments labelled $r_0 \ldots r_6$ which are controlled individually in order to create a suitable image: in our case we deal only with images related to decimal digits $0 \ldots 9$. We need $n = 4$ bits to represent a decimal digit. Therefore a decoder to translate from a digit to the corresponding image will have a 4-bit input and $2^4 = 16$ outputs; an encoder to perform the reverse translation from an image to the corresponding digit will have a 16-bit input and 4-bit output.

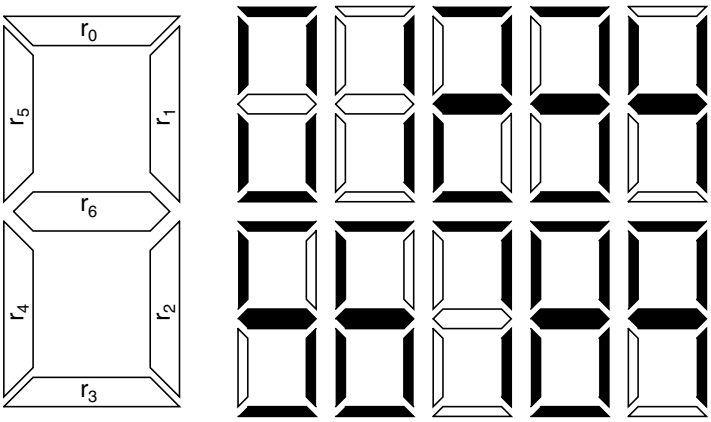


Figure 2.21 A 7-segment LED.

i_3	i_2	i_1	i_0	r_6	r_5	r_4	r_3	r_2	r_1	r_0
0	0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	1	1	0
0	0	1	0	1	0	1	1	0	1	1
0	0	1	1	1	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1	0
0	1	0	1	1	1	0	1	1	0	1
0	1	1	0	1	1	1	1	1	0	1
0	1	1	1	0	0	0	0	1	1	1
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	0	1	1	1

Table 2.14 Truth table for a 7-segment LED decoder.

Table 2.14 describes a truth table for a decoder which could be used to control the LED. It leaves out unused input and output combinations for brevity. For example, we do not use the possible digits $10 \ldots 15$ and some of the potential segment numbers are not used; there are no segments $r_7 \ldots r_{15}$. In reality, these relate to don't care states. To implement the decoder, one can derive a logical expression for each of the outputs via a Karnaugh map or similar technique

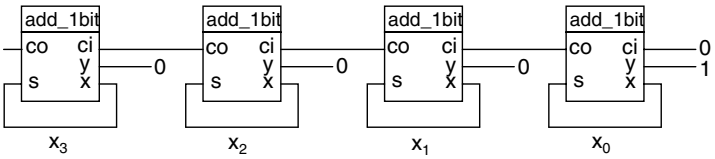


Figure 2.22 A flawed, nonsensical example of a 4-bit counter where we neither initialise the value *x or* allow it to settle before incrementing it.

$$
\begin{array}{llll}
r_0 = i_3 & \vee\, i_1 & \vee\, (i_0 \wedge i_2) & \vee\, (\neg i_0 \wedge \neg i_2) \\
r_1 = i_3 & \vee\, \neg i_2 & \vee\, (i_0 \wedge i_1) & \vee\, (\neg i_0 \wedge \neg i_1) \\
r_2 = i_3 & \vee\, \neg i_1 & \vee\, (\neg i_3 \wedge i_2) & \vee\, (i_0 \wedge i_1) \\
r_3 = (\neg i_0 \wedge \neg i_2) & \vee\, (\neg i_0 \wedge i_1) & \vee\, (i_0 \wedge \neg i_1 \wedge i_2) & \vee\, (i_0 \wedge i_1 \wedge \neg i_2) \\
r_4 = (\neg i_0 \wedge \neg i_2) & \vee\, (\neg i_0 \wedge i_1) & & \\
r_5 = i_3 & \vee\, (\neg i_0 \wedge \neg i_1) & \vee\, (\neg i_1 \wedge i_2) & \vee\, (\neg i_0 \wedge i_2) \\
r_6 = (\neg i_0 \wedge i_1) & \vee\, (\neg i_2 \wedge i_3) & \vee\, (\neg i_1 \wedge i_2) & \vee\, (i_1 \wedge \neg i_2 \wedge \neg i_3)
\end{array}
$$

Implementation of the decoder is then simply implementation of the expressions with suitable sharing of common terms to reduce the number of gates required.

2.3 Clocked and Stateful Logic

Combinatorial logic is continuous in the sense that the output is always being computed, subject to propagation delay, as a product of the inputs. One of the limitations of this sort of component is that it cannot store state, it has no memory. As an example of why this could be an issue, consider the problem of implementing a counter that repeatedly computes $x \leftarrow x + 1$ for some 4-bit value x. That is, it takes x, increments it and stores the result back into the value x ready to start again. Given we already have a 1-bit adder component, a naive first attempt at realising such an operation might be to build the component shown in Figure 2.22. Essentially we take the value x, feed it into the adder and loop the adder output back into the input ready to start again.

There are two *major* problems with this approach. Firstly, we have not given x and initial value so it is unclear what the input to the adder will be when we start. Secondly and more importantly, we never actually store the value x. The value fed to the input of the adder is continuously being updated by the output; as the output changes the input changes and so the output changes again, the circuit is never able to settle into a consistent state. These problems highlight a big difference between designing hardware and writing software. In a language such as C, we are given variables which can store or remember values for us; sequences of statements are guaranteed to execute in order with each only being executed once previous ones

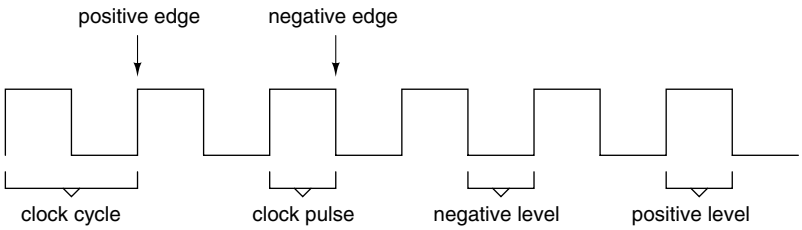


Figure 2.23 Features of a clock signal.

have completed. This section aims to address this imbalance by showing how we can build components to store values and how to perform sequences of operations in a controlled manner.

2.3.1 Clocks

In order to execute a sequence of operations correctly, we need some means of controlling and synchronising them. The concept of a **clock** is central to achieving this. When we say clock in digital circuit design we usually mean a signal that oscillates between 0 and 1 in a regular fashion rather than something that tells the time. This sort of clock is more like a metronome; ticks of the clock act as synchronising events that keep other components in step with each other.

A **clock signal** is just the same as any other value within a logic design. As such, there are several items of terminology that relate to general values but are best introduced in the context of clocks. Figure 2.23 details these features. Firstly, we assume the clock signal approximates a square wave so the signal is either 0 or 1. The physical characteristics of these components mean this is dubious in reality (the corners of the signal can be "rounded"), but suffices for the discussion here. Transitions of the clock from 0 to 1 are called positive **clock edges**, transitions from 1 to 0 are negative clock edges. At any point while the clock is 1 we say it is at a positive **level**, while the clock is 0 it is at a negative level. The interval between one positive or negative edge and the next positive or negative edge respectively is termed a **clock cycle**; the time taken for one clock cycle to happen is the **clock period** and the number of clock cycles that occur each second is the **clock frequency** or **clock rate**. The region between a positive and negative clock edge is termed a **clock pulse**. Note that the time the clock signal spends at positive and negative levels need not be equal; the term **duty cycle** is used to describe the ratio between these times. The clocks we use will typically have a duty cycle of a half meaning the signal is at a positive level for the same time it is at a negative level.

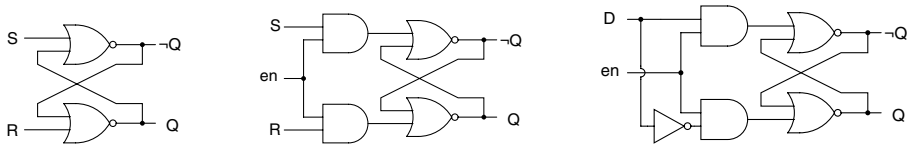
The basic idea in using clocks to synchronise events within a circuit is that on a clock edge we present values to our combinatorial logic in order to do some com-

C_3	C_2	C_1	C_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Table 2.15 Values of bits in a 4-bit counter as used for clock division.

putation with them. These values are only updated on clock edges, as a result the combinatorial logic has one clock cycle to settle into a stable state; note that this differs from the flawed example above where values are updated instantaneously. As such, the faster our clock frequency is, the faster we perform computation. However, there are two limits to how a given circuit can be clocked. Firstly, since the combinatorial logic must compute a new result within one clock cycle, the clock period must be greater than the critical path of that logic. Recall that the critical path dominates the time taken to compute a result, if it is longer than the clock period then the result required to update our value with will not be ready. This is a crucial fact: to perform computation at high speeds it is vital that the critical path of the combinatorial components is minimised so the clock frequency can be maximised. Secondly, we must consider the effects of propagation delay in wires to prevent a phenomenon called **clock skew**. Clock skew is simply the fact that if a clock signal arrives at one point along a slightly longer wire than another point, the times of arrival will differ slightly; the clocks will be skewed or unsynchronised. Depending on the situation this might present a problem because the circuit components are no longer perfectly in step.

Although the clock is just any other value, we need it to oscillate in a regular and reliable manner. As such, clock signals are usually generated externally to a given circuit and fed into it. A common method for generating the required behaviour is to use a piezoelectric crystal which, when a voltage is applied across it, oscillates according to a natural frequency related to the physical characteristics of the crys-



(a) SR-latch. (b) Clocked SR-latch. (c) D-type latch.

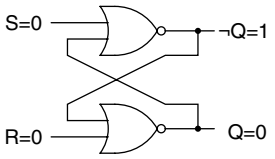
Figure 2.24 Three variations of SR-latch.

tal. Roughly speaking, one can use the resulting electrical field generated by this oscillation as the clock signal.

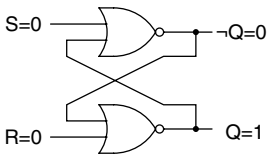
Multiplying a reference clock, or increasing the frequency, is best done using a dedicated component and is somewhat beyond the scope of discussion here. Dividing a reference clock, or decreasing the frequency, is achieved much more easily. Imagine that on each positive edge of our original clock, a 4-bit counter C is incremented. Table 2.15 shows the progression of the counter with row i representing the value of bits in C after i clock edges have passed. The value of bit C_0 oscillates from 0 to 1 at half the same speed as the original clock; the value of bit C_1 takes twice as long again, it oscillates from 0 to 1 at a quarter of the speed as the original clock. Thus by using our counter and examining specific bits, we have created a component which can divide the clock frequency by powers-of-two. Specifically, if we want to divide the clock by 2^j (with $j > 0$), we use a j-bit counter and examine the $(j-1)$-th bit.

2.3.2 Latches

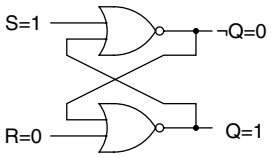
The first step toward building a component which can remember or store values is somewhat counter-intuitive. We start by looking at the Set-Reset latch, usually called an **SR-latch**, as shown in Figure 2.24a. The circuit has two inputs called S and R which are the set and reset signals, and two outputs Q and $\neg Q$ which are always the inverse of each other. The circuit design seems a little odd compared to what we have seen so far because we have introduced a sort of loop; the outputs of each NOR gate are wired to the input of the other. The result of this odd design is that the outputs of the circuit are not uniquely defined by the inputs. For example, when we set the inputs to $S = 0$ and $R = 0$, either of the two situations in Figure 2.25a-b is valid; the circuit is in a stable state in both cases. When we set the inputs to $S = 1$ and $R = 0$, we force the latch into the state shown in Figure 2.25c; Q is set to 1 and $\neg Q$ to 0 whatever they were before because the top NOR gate will have to output 0. Conversely, when we set the inputs to $S = 0$ and $R = 1$ we force the latch into the state shown in Figure 2.25d; Q is set to 0 and $\neg Q$ to 1 because the bottom NOR gate



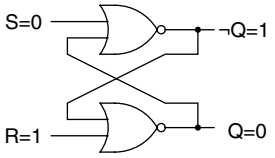
(a) $S = 0, R = 0$.



(b) $S = 0, R = 0$.



(c) $S = 1, R = 0$.



(d) $S = 0, R = 1$.

Figure 2.25 Stable states for the SR-latch.

		current		*next*	
S	R	Q	$\neg Q$	Q	$\neg Q$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	?	?	?	?

(a) SR-latch

		current		*next*	
J	K	Q	$\neg Q$	Q	$\neg Q$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	0	1	1	0
1	1	1	0	0	1

(b) JK-latch

Table 2.16 Truth tables for an SR-latch and a JK-latch.

will have to output 0. The final issue is what happens if we set the inputs to $S = 1$ and $R = 1$. In this case, the circuit enters a contradictory or undefined state: both outputs should become 0, yet we know they must also be the inverse of each other.

Defining the metastable state $S = 1, R = 1$ as unknown, the behaviour of the SR-latch can be described by Table 2.16a. Note that this differs somewhat from previous truth tables because there is some notion of state in Q and $\neg Q$. Their value when they are next latched depends partly on their current value. Essentially we have a component that we can set into a defined state using the set and reset inputs: if we want the output Q to be 1 we set $S = 1, R = 0$ for a while; if we want the output to be 0 we set $S = 0, R = 1$ for a while. The key thing to notice is that after waiting a while, when we return the inputs to $S = 0, R = 0$ the circuit remembers which of S

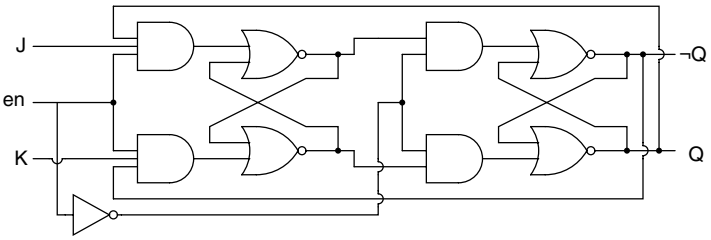


Figure 2.26 Implementation of a master-slave JK-latch.

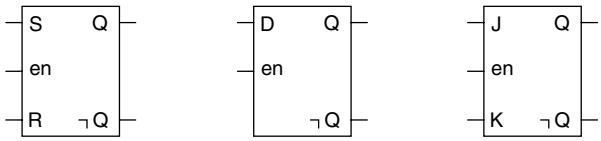
and R was set last. It offers a means of storing or latching a 1-bit value provided we control it correctly.

One problem with the basic SR-latch design is that it cannot currently be synchronised with other components in the system, i.e., the latched value can be changed at *any* time. It is often convenient to constrain when the value can be changed; as a result we introduce the variation shown in Figure 2.24b. The basic idea is to AND an enable signal *en* with the S and R inputs to provide the latch inputs. By doing so, the latch is only sensitive to changes in S and R during positive levels of *en*; when $en = 0$ it does not matter what S and R are, the AND gates simply output 0. Since *en* will often be a clock signal, but is not required to be, we call this component a **clocked SR-latch**. In this context, the latched value can only change value during the positive phase of a clock cycle.

As a side note, we usually call components which are sensitive to inputs during positive or negative levels of an enable signal **active high** and **active low** respectively. The choice of which to use is usually dictated by cost of implementation and ease of integration with other components: there is no real advantage in using one over the other from any other point of view.

Using either an active high or active low approach, a further problem with our clocked SR-latch is the potential to enter an unattractive, metastable state when $S = 1$ and $R = 1$. The easiest way to avoid this situation is to ensure it never occurs; we can do this by allowing one input D and using $S = D$ and $R = \neg D$ as inputs to a clocked latch. This approach is shown in Figure 2.24c. Clearly we can now never have that $S = 1$ while $R = 1$ and so avoid the potential problem. However, we can also not have that $S = 0$ while $R = 0$ which was the state which remembered the input. This design, termed a **D-type latch**, is a component which operates such that when $D = 1$ and $en = 1$ the output $Q = 1$, while if $D = 0$ and $en = 1$ the output $Q = 0$. If $en = 0$ the D-type latch simply maintains the current state. We now have a reliable 1-bit memory. When we want to latch a new value we place it on the input D and send a clock pulse into *en*; the currently stored value is always available on the output Q.

An alternative approach to solving the metastable state problem is to use the input $S = 1$, $R = 1$ for some other purpose. The **JK-latch** is an example of this



(a) SR-latch. (b) D-type latch. (c) JK-latch.

Figure 2.27 Symbolic representations of different latches.

which uses the state to toggle the outputs Q and $\neg Q$; when $S = 1$ and $R = 1$ the outputs swap over as described in Table 2.16b. Although other methods exist, the classic implementation of a JK-latch uses two SR-latches in series: the first is called the master while the second is called the slave, an arrangement termed a master-slave JK-latch and is shown in Figure 2.26. By inverting *en* and feeding the result to the slave SR-latch, we enable the master SR-latch during positive levels of the enable signal while disabling the slave. When the enable signal changes, the value held in the master SR-latch is transferred to the slave. Roughly speaking, since there is no longer a feedback path directly from one latch into itself, the metastable state is eliminated. Or, by feeding Q and $\neg Q$ back into the input AND gates we can never feed $S = 1$ and $R = 1$ because at least one of Q and $\neg Q$ will be 0 in the slave latch.

We typically use the symbols in Figure 2.27 to represent the different latches presented here so as to simplify the description of large designs. It is common to further simplify the symbols by omitting $\neg Q$ unless it is explicitly required.

2.3.3 Flip-Flops

Latches are **level triggered** in the sense that their value can be changed when the enable signal is at a positive or negative level depending on whether then are active high or low. This partly solves the problem of synchronisation but it is often attractive to constrain things even further and say that the value should only be able to change at a positive or negative *edge* rather than at any time during the positive or negative *level*. Altering our basic latch component so that it is **edge triggered** produces a variation called a **flip-flop**.

A basic approach for changing a latch into a flip-flop is to construct a pulse generator circuit which remains at a high-level for a very short period of time after the enable signal goes high. We use this pulse generator to drive a level triggered latch; the reasoning is that if the pulse is short enough, the active period of the latch is essentially limited to the positive edge of the enable signal. Figure 2.28 shows an implementation of such a device while Figure 2.29 details the resulting behaviour assuming propagation delays for NOT and AND gates are again 10ns and 20ns respectively. Propagation in the NOT gate is what makes the circuit work, when *en*

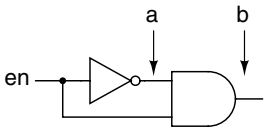


Figure 2.28 Implementation of a simple pulse generator.

is set to 1 at time $t = 10$ns there is a 10ns delay before a changes. This delay means that for a short time equal to the propagation delay of the NOT gate, both *en* and a are equal to 1 meaning that b is also 1. The change in b is delayed by 20ns but the short pulse we are looking for is still evident; the pulse width is equal to the propagation delay of the NOT gate, it is skewed by a distance equal to the propagation delay of the AND gate. Notice that when *en* is set to 0 again at time $t = 70$ns we do not get a similar pulse: this generator produces positive edge triggered devices when coupled to it. Thus, as long as the propagation delay of the NOT gate is small enough, we can alter, for example, a standard D-type latch to produce an edge triggered D-type flip-flop shown in Figure 2.30.

We again use symbols, shown in Figure 2.31, to represent the different flip-flop types. The only difference from the symbol for the corresponding latch is the inclusion of a vee-shaped marker on the *en* input which denotes edge triggering rather than level triggering.

2.3.4 State Machines

Definition 31. A **Finite State Machine (FSM)** is formally defined by a number of parts:

- Σ, an input alphabet.
- Q, a finite set of states.
- $q_0 \in Q$, a start state.
- $A \subseteq Q$, a set of accepting states.
- A function $\delta : Q \times \Sigma \to Q$ which we call the transition function.

More simply an FSM is just a directed graph where moving between nodes (which represent each state) means consuming the input on the corresponding edge. As an example, consider a simple state machine that describes a vending machine that sells chocolate bars. The machine accepts tokens worth 10 or 20 units but does not give change. When the total value of tokens entered reaches 30 units it delivers a chocolate bar but it does not give change; the exact amount must be entered otherwise an error occurs, all tokens are ejected and we start afresh.

From this description we can start defining the state machine components. The input alphabet is simply the possible input values, in this case the different tokens

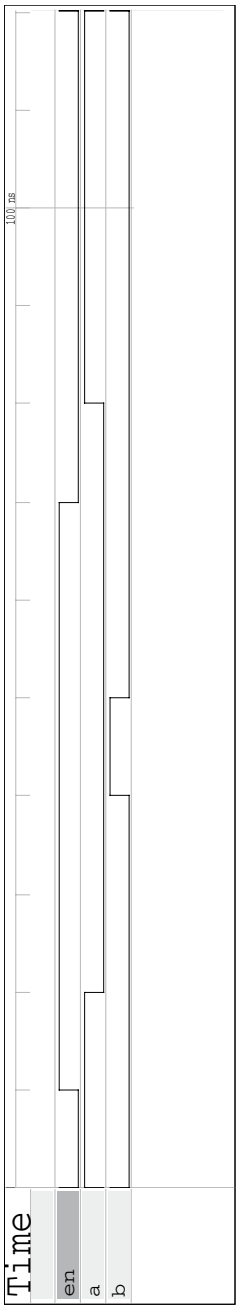


Figure 2.29 Behavioural time-line for a simple pulse generator.

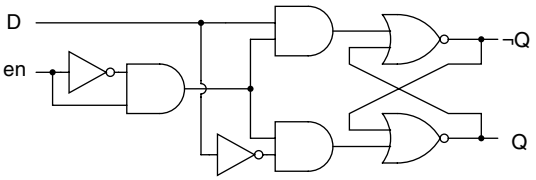
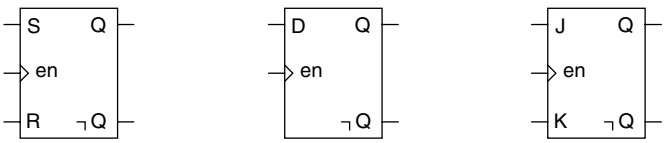


Figure 2.30 Implementation of a D-type flip-flop.



(a) SR flip-flop. (b) D-type flip-flop. (c) JK flip-flop.

Figure 2.31 Symbolic representations of different flip-flops.

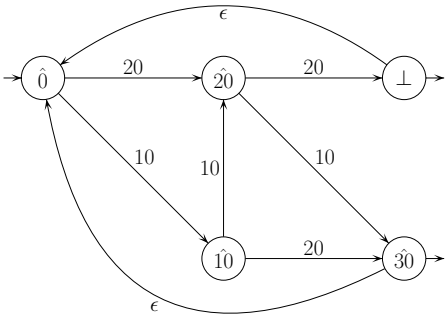


Figure 2.32 A graph describing the FSM for a chocolate vending machine.

such that $\Sigma = \{10, 20\}$. The set of states the machine can be in is easy to imagine: it can either have tokens totalling 0, 10, 20 or 30 units in it or be in the error state which we denote by $\perp$. Thus, using a hat over numbers to differentiate between states and input token values, we have that $Q = \{\hat{0}, \hat{10}, \hat{20}, \hat{30}, \perp\}$ where $q_0 = \hat{0}$ since we start with 0 tokens in the machine. The accepting state is $\hat{30}$ since this represents the state where the user has entered a valid sequence of tokens and obtains a chocolate bar. Finally we need to define the transition function δ which is best described diagrammatically as in Figure 2.32. From this we see that, for example, if we are in state $\hat{20}$ and accept a 10 unit token we end up in state $\hat{30}$, while if we accept a 20 unit token we end up in the error state. Note that the input marked ε is the empty

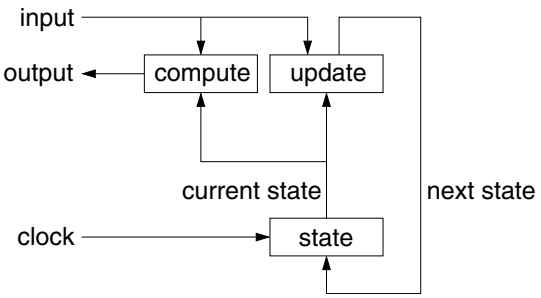


Figure 2.33 The general structure of an FSM in digital logic.

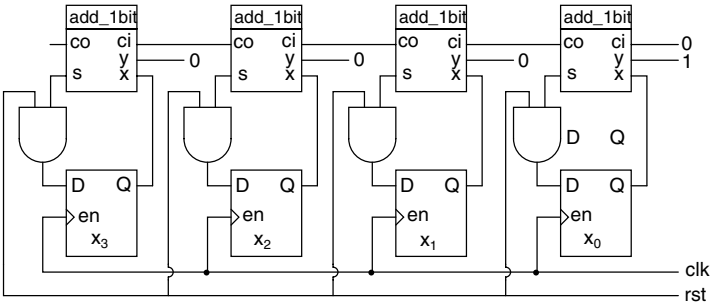


Figure 2.34 A corrected example of a 4-bit counter.

input; that is, with no input we can move between the accepting or error states back into the start state thus resetting the machine.

From this description it should start to become obvious why state machines play a big part in digital logic designs. Given we now have components that allow us to remember some state, as long as we can construct combinatorial logic components to compute the transition function δ, and a clock to synchronise components, we can think of a design that allows us to step through sequences of operations. A general structure for digital logic state machines is shown in Figure 2.33. A block of edge triggered flip-flops is used to store the current state; this is fed into two combinatorial components whose task it is to update the state by applying the transition function and compute any outputs based on the state we are in. On each positive clock edge our flip-flops take the next state and store it, overwriting the current state. Thus the design steps into a new state under control of the clock.

Using this general structure we can finally solve the problem outlined at the beginning of the section: how to build a 4-bit counter. Clearly we need four flip-flops to store the 4-bit state of our counter; these constitute the state portion of the general structure. We do not require any additional output from the circuit, simply that it cy-

State	Next State	Main Road	Annex Road
0	1	green	red
1	2	amber	red
2	3	red	amber
3	4	red	green
4	5	red	amber
5	0	amber	red

Table 2.17 Description of the traffic light controller state machine.

S_2	S_1	S_0	S'_2	S'_1	S'_0	M_r	M_a	M_g	A_r	A_a	A_g
0	0	0	0	0	1	0	0	1	1	0	0
0	0	1	0	1	0	0	1	0	1	0	0
0	1	0	0	1	1	1	0	0	0	1	0
0	1	1	1	0	0	1	0	0	0	0	1
1	0	0	1	0	1	1	0	0	0	1	0
1	0	1	0	0	0	0	1	0	1	0	0

Table 2.18 Truth table for the traffic light controller state machine.

cles through values in turn. Thus we only need a simple update portion built from a 4-bit adder as before. Figure 2.34 demonstrates the final design. When the reset signal *rst* is 0, the counter is set to 0 on the following positive clock edge. When the reset signal is 1, the counter is updated on each positive clock edge by feeding the current value through the adder. Since there is a whole clock cycle between presenting the current value and storing the new value, the adder can settle in a stable state and we get the correct result.

As a more complete example of state machines in this context, consider the problem of controlling a set of traffic lights which prevent motorists on a main road crashing into those turning on to it from an annex road. The idea is that while traffic is set flowing on the main road via a green light or go sign, the annex road drivers should be shown a red light or stop sign. Eventually drivers on the annex road get a turn and the roles are reversed. An amber light is added to the system to offer a period of changeover where both roads are warned that the lights are changing.

Table 2.17 offers a description of the problem. It describes a state machine with six states and, given a current state, which state we should move into next: there is no input in this machine, each of the transitions in the function δ being ε type and based only on the state we are currently in. Furthermore, it shows what lights should be on given the current state. So if we are in state 0, for example, we should move to state 1 next and the main and annex road lights should be green and red respectively. We could draw a graph of the state machine but it would be a bit boring since it simply cycles through the six inputs in sequence.

The states can be represented as 3-bit integers since there are six states and $2^3 > 6$. We use S_0, S_1 and S_2 to denote the current state bits and S'_0, S'_1 and S'_2 to denote the next state bits. There are nine outputs from our combinatorial logic, three outputs for the next state and six outputs to represent the red, amber and green lights on each of the roads. We use M_r, M_a and M_g to denote red, amber and green lights on the main road and A_r, A_a and A_g to denote red, amber and green lights on the annex road.

As a side note, the encoding of our states is important in some cases; we have simply used the binary representation of each state number but more involved schemes exist. For example, one could use a Gray code, mentioned earlier when we introduced Karnaugh maps. Using a Gray code to represent states means that only one bit changes between adjacent states: this might reduce transistor switching and thus improve the power efficiency of our design. Alternatively one can consider a technique called **one-hot coding**. Essentially this uses one flip-flop for each state; when the i-th flip-flop stores 1, we are in state i. The state machine is reset by setting all flip-flops to 0 except the first flip-flop which is set to 1 to store the hot bit. The flip-flops are chained together so that on each clock edge the hot bit is passed to the next flip-flop and we transfer from state to state. One advantage of this approach is that decoding which state we are in is made much easier; one disadvantage is that we typically use more logic to store the state.

From this tabular description of the traffic light problem we can derive a truth table, shown in Table 2.18, for each of our outputs given the current state. Note that for each light we assume a signal of 1 turns the light on while a signal of 0 turns it off. Using this truth table we can derive expressions for the next state in terms of the current state:

$$\begin{aligned}
S'_0 &= (\ \neg S_0 \qquad\qquad\qquad\ ) \\
S'_1 &= (\ \ \ S_0 \wedge \neg S_1 \wedge \neg S_2\ ) \vee \\
&\quad\ (\ \neg S_0 \wedge \ \ S_1 \wedge \neg S_2\ ) \\
S'_2 &= (\ \ \ S_0 \wedge \ \ S_1 \wedge \neg S_2\ ) \vee \\
&\quad\ (\ \neg S_0 \wedge \neg S_1 \wedge \ \ S_2\ )\ .
\end{aligned}$$

Note that if our state machine transition function depended on some input as well as the current state, our expressions above would have to include that input in order to produce the correct next state given the current one. Either way, we can do a similar thing to control the traffic light colours on each road:

$$
\begin{aligned}
M_r &= (\ \neg S_0 \wedge \ \ S_1 \wedge \neg S_2\ ) \vee \\
&\quad (\ \ S_0 \wedge \ \ S_1 \wedge \neg S_2\ ) \vee \\
&\quad (\ \neg S_0 \wedge \neg S_1 \wedge \ \ S_2\ ) \\
M_a &= (\ \ S_0 \wedge \neg S_1 \wedge \neg S_2\ ) \vee \\
&\quad (\ \ S_0 \wedge \neg S_1 \wedge \ \ S_2\ ) \\
M_g &= (\ \neg S_0 \wedge \neg S_1 \wedge \neg S_2\ ) \\
\\
A_r &= (\ \neg S_0 \wedge \neg S_1 \wedge \neg S_2\ ) \vee \\
&\quad (\ \ S_0 \wedge \neg S_1 \wedge \neg S_2\ ) \vee \\
&\quad (\ \ S_0 \wedge \neg S_1 \wedge \ \ S_2\ ) \\
A_a &= (\ \neg S_0 \wedge \ \ S_1 \wedge \neg S_2\ ) \vee \\
&\quad (\ \neg S_0 \wedge \neg S_1 \wedge \ \ S_2\ ) \\
A_g &= (\ \ S_0 \wedge \ \ S_1 \wedge \neg S_2\ )\ .
\end{aligned}
$$

Thus the current state drives everything; we feed S_0, S_1 and S_2 into two components, one of which produces the next state we need to move into and one which produces signals to control the lights. Clearly by sharing terms between the state update and light components we can reduce the cost of our design significantly.

2.4 Implementation and Fabrication Technologies

Designing a digital circuit is roughly akin to writing a program: once we have written a program, one usually intends to use it to perform some real task by executing it on a computer. The same is true of circuits, except that we first need to realise them somehow using physical components in order to use them. Although this topic is somewhat beyond the scope of this book, it is useful to understand mechanisms by which this is possible: even if that understanding is at a high-level, it can help to connect theoretical concepts with their practical realisation.

2.4.1 Silicon Fabrication

2.4.1.1 Lithography

The construction of semiconductor-based circuitry is very similar to how pictures are printed; essentially it involves controlled use of chemical processes. The act of printing pictures onto a surface is termed **lithography** and has been used for a couple of centuries to produce posters, maps and so on. The basic idea is to coat the surface, which we usually call the **substrate**, with a photosensitive chemical. We then expose the substrate to light projected through a negative, or **mask**, of the required image; the end result is that a representation of the original image is left on the substrate where the light reacts with the photosensitive chemical. After

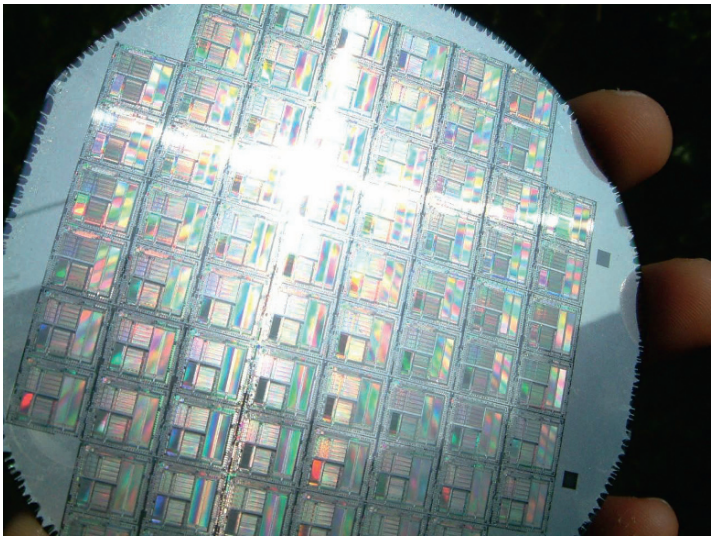


Figure 2.35 A silicon wafer housing many instances of an INMOS T400 processor design; see finger tips for scale ! (Copyright remains with photographer, photographer: James Irwin, source: personal communication)

washing the substrate, one can treat it with further chemicals so that the treated areas representing the original image are able to accept inks while the rest of the substrate cannot.

The analogous term in the context of semiconductors is **photo-lithography** and roughly involves similar steps. We again start with a substrate, which is usually a wafer of silicon. The wafer is circular because of the process of machining it from a synthetic ingot, or boule, of very pure silicon. After being cut into shape, the wafer is polished to produce a surface suitable for the next stage. We can now coat it with a layer of base material we wish to work with, for example a doped silicon or metal.

We then coat the whole thing with a photosensitive chemical, usually called a **photo-resist**. Two types exist, a positive one which hardens when hidden from light and a negative one which hardens when exposed to light. By projecting a mask of the circuit onto the result, one can harden the photo-resist so that only the required areas are covered with a hardened covering. After baking the result to fix the hardened photo-resist, and **etching** to remove the surplus base material, one is left with a layer of the base material only where dictated by the mask. A further stage of etching removes the hardened photo-resist completes the process for this layer. Many layers are required to produce the different layers required in a typical circuit; for example, we might need layers of N-type and P-type semiconductor and a metal layer to produce transistors. Using photo-lithography to produce a CMOS-based circuit, for example, thus requires many rounds of processing.

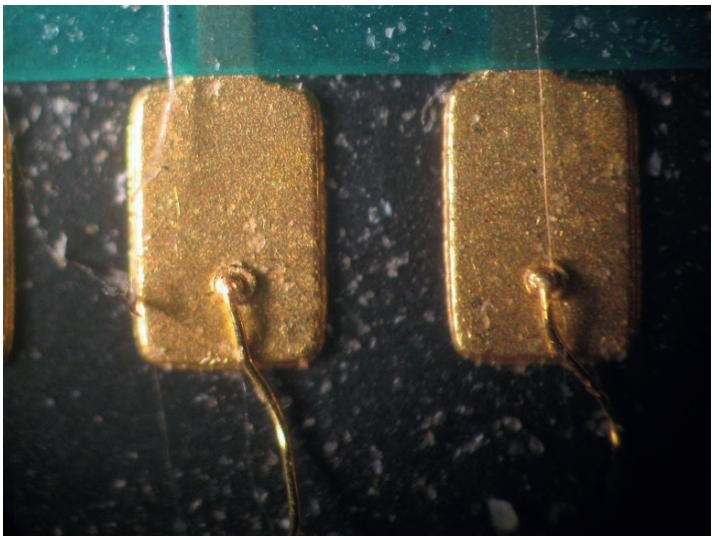


Figure 2.36 Bonding wires connected to a high quality gold pad. (Public domain image, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:Wirebond-ballbond.jpg`)

2.4.1.2 Packaging

A major advantage of lithography is that one can produce accurate results across the whole wafer in one go. As such, one would typically produce many individual but identical circuits on the same wafer; each circuit instance is called a **die**. The next step in manufacture is to subject the wafer to a series of tests to ensure the fabrication process has been successful. Some dies will naturally be defective due to the circular shape of the wafer and typically rectangular shape of the dies; some dies will only represent half a circuit ! More importantly, since the manufacturing process is imperfect even complete dies can be operationally defective. Some circuit designs will incorporate dedicated test circuitry to aid this process. The ratio of intended dies to those that actually work is called the **yield**; clearly a high yield is attractive since defective circuits represent lost income.

The final step is to cut out each die from the wafer and **package** the working components. Packaging involves mounting the silicon circuit on a plastic base and connecting the tiny circuit inputs and outputs to externally accessible **pins** or **pads** with high quality **bonding wires**. The external pins allow the circuit to be interfaced with the outside world, the final component can thus be integrated within a larger design and forms what we commonly call a **microchip** or simply a **chip**. Large or power-hungry microchips are often complemented with a heat sink and fan which

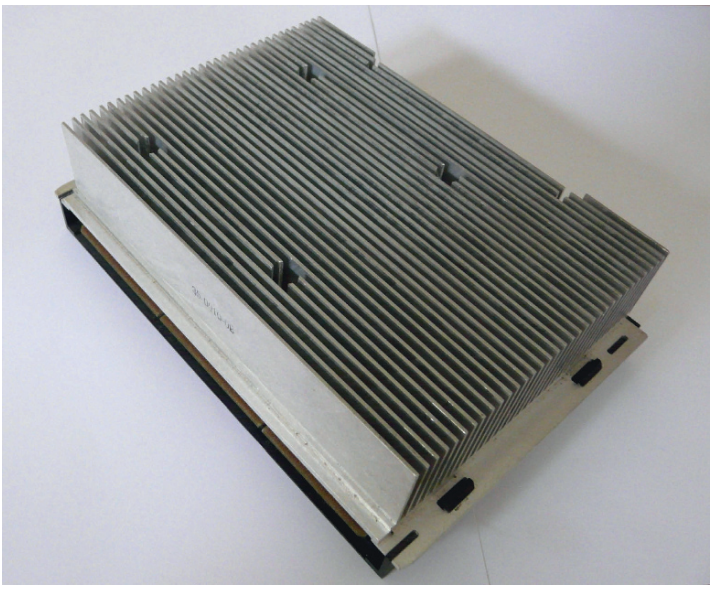


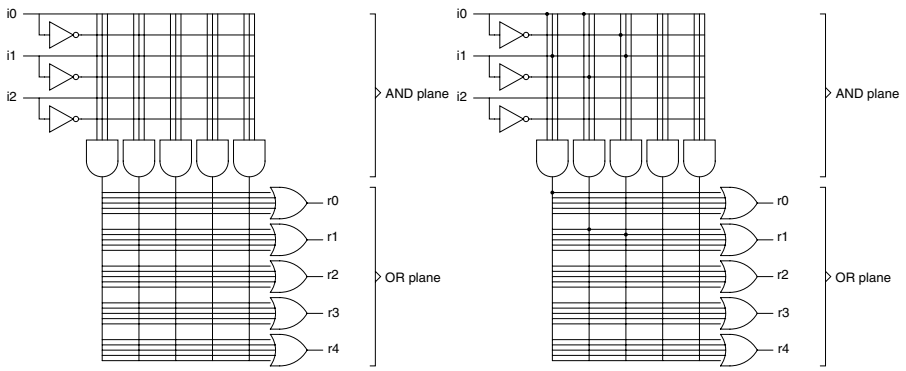
Figure 2.37 A massive heatsink (roughly 8" × 4" × 1.5") sat on top of a 667 MHz DEC Alpha EV67 (21264A) processor; such aggressive cooling allowed the processor to be run at higher clock speeds than suggested ! (Copyright remains with photographer, photographer: James Irwin, source: personal communication)

draws heat away and allow it to more easily operate within physical limits of the constituent materials.

2.4.2 Programmable Logic Arrays

A **Programmable Logic Array (PLA)** allows one to implement most combinatorial logic functions from a single, generic device. The basic idea is that the PLA forms a network of AND and OR gates. Considering an AND-OR type PLA as an example, the first network, or **plane**, allows one to implement minterms using a set of AND gates. These minterms are fed into a second plane of OR gates which implement the SoP expression for the required function. An OR-AND type PLA reverses the ordering of the planes of gates and thus allows implementation of PoS expressions.

More than one function that uses the same set of inputs can be implemented by the same PLA in either case. However, the limit in terms of both numbers of input and output and design complexity, is governed by the number of gates and wires one can fit into each plane. Figure 2.38a shows a diagram of a clean AND-OR type PLA



(a) A clean PLA. (b) A PLA configured as a half-adder.

Figure 2.38 Implementations of a clean and a configured PLA.

device with three inputs and five outputs. Notice how we construct the inverse of each input and feed wires across the plane so that they can be connected to produce the required logic function.

To program or configure a PLA so that it computes the required function, one starts with a generic, clean device with no connections on the wires. For an AND-OR type device, connection of the wires, by filling in the black dots on the diagram, forms the minterms in the AND plane and collects them in the OR plane. From a physical perspective, this is done by manipulating components placed at the connection points. Consider the placement of a fuse at each connection point or junction. Normally this component would act as a conductive material somewhat like a wire; when the fuse is blown using some directed energy, it becomes a resistive material. Thus, to form the right connections on a PLA one simply blows all the fuses where no connection is required. Alternatively, one can consider an antifuse which acts in the opposite way to a fuse: normally it is a resistor but when blown it is a conductor. Using antifuses at each junction means the configuration can simply blow those antifuses at the junctions where a connection is required. Figure 2.38b shows the originally clean PLA configured to act as a half-adder. We have that i_0 and i_1 are the inputs while r_0 is the carry-out and r_1 is the sum.

PLA devices based on fuses and antifuses are termed one-time-programmable since once the components at each junction are blown, they cannot be unblown: the device can only be configured once. However, the advantage of a PLA is the generality it offers. That is, one can take a single PLA component and easily implement most combinatorial functions: no costly silicon fabrication is required.

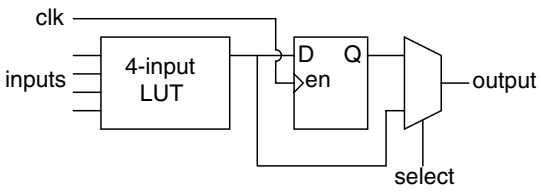


Figure 2.39 Implementation of a basic FPGA logic block.

2.4.3 Field Programmable Gate Arrays

The problems of one-time-programmable PLA devices are clear: prototyping logic designs can be costly since any mistake is terminal and the device can only ever be used for one task which, for large designs, means more cost when redundant devices are left idle. A **Field Programmable Gate Array (FPGA)** is one type of device that acts to solve this problem. Essentially, the idea of an FPGA is to offer a many-time-programmable device; the FPGA can be configured with one design and then re-configured with another design at a later date.

Roughly, an FPGA is built from a mesh of logic blocks. Each block can be configured to perform a specific, although limited, range of logic functions. Typical logic blocks might contain an n-input, 1-output **Look-Up Table (LUT)**. The LUT operation is simple: one forms an integer index i from the n inputs, looks-up item i in a table and returns this as the result. By filling the table with the required outputs for each input, the LUT can implement any n-input, 1-output logic function. Along side each LUT we typically find one or more flip-flops to enable devices with memory to be constructed. Most FPGAs also offer a small number of specialist logic blocks which incorporate high-performance versions of common circuits such as adders or even small processors. Figure 2.39 details a simple logic block consisting of a 4-input LUT and a D-type flip-flop; this is typical of many FPGA architectures.

Connections between the logic blocks in the mesh are formed in a similar way to a PLA in the sense that a component is placed at each connection junction. However, the FPGA junctions are re-configurable switches that can be turned on and off as required rather than blown in a one-off act. The idea is that by configuring each logic block to perform the right task and configuring connections between them, one can have the mesh compute both combinatorial and stateful logic circuits. For example, with our simple logic block above the configuration would fill the LUT with correct values and initialise the select input of the multiplexer with the right value so the block performs the right role. Connections between the logic blocks are then configured so that they interconnect and form the required circuit.

FPGA devices typically operate more slowly than custom silicon and use more power. As such, they are often used as a prototyping device for designs which will eventually be fabricated using a more high-performance technology. Further, in applications where debugging and updating hardware is as essential as software the

FPGA can offer significant advantages. Use of FPGA devices in mission-critical applications such as space exploration is commonplace: it turns out to be exceptionally useful to be able to remotely fix bugs in ones hardware rather than write off a multi-million pound satellite which is orbiting Mars and hence out of the reach of any local repair men !

2.5 Further Reading

- T.L. Floyd.
  Digital Fundamentals.
  Prentice-Hall, 2005. ISBN: 0-131-97255-3.
- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.
- C. Roth.
  Fundamentals of Logic Design.
  Brooks-Cole, 2003. ISBN: 0-534-37804-8.
- A.S. Tanenbaum.
  Structured Computer Organisation.
  Prentice-Hall, 2005. ISBN: 0-131-48521-0.

2.6 Example Questions

5. a. Describe how *N*-type and *P*-type MOSFET transistors are constructed using silicon and how they operate as switches.

b. Draw a diagram to show how *N*-type and *P*-type MOSFET transistors can be used to implement a NAND gate. Show your design works by describing the transistor states for each input combination.

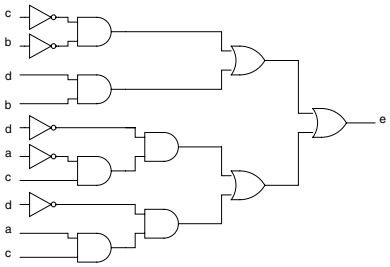
6. Given that ? is the don't care state, consider the following truth table which describes a function p with four inputs (a, b, c and d) and two outputs (e and f):

a	b	c	d	e	f
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	?	?
0	1	0	0	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	?	?

a	b	c	d	e	f
1	0	0	0	1	0
1	0	0	1	0	0
1	0	1	0	0	1
1	0	1	1	?	?
1	1	0	0	?	?
1	1	0	1	?	?
1	1	1	0	?	?
1	1	1	1	?	?

a. From the truth table above, write down the corresponding sum of products (SoP) equations for e and f.
b. Simplify the two SoP equations so that they use the minimum number of logic gates possible. You can assume the two equations can share logic.

7. Consider the following circuit where the propagation delay of logic gates in the circuit are NOT= 10 ns; AND= 20 ns; OR= 20 ns; XOR= 60 ns.



a. Draw a Karnaugh map for this circuit and derive a sum of products (SoP) expression for the result.
b. Describe advantages and disadvantages of your SoP expression and the dynamic behaviour it produces.
c. If the circuit is used as combinatorial logic within a clocked system, what is the maximum clock speed of the system ?

8. You are tasked with building a device for a casino based gambling game. The device must check when three consecutive heads are generated by a random coin flipper. Design a hardware circuit which will output 1 when three consecutive heads are generated and 0 otherwise. You can assume the random coin flipper circuit is already built or, for extra marks, design one yourself.

9. Imagine you are asked to to build a simple DNA matching hardware circuit as part of a research project. The circuit will be given DNA strings which are sequences of tokens that represent chemical building blocks. The goal is to search a large input sequence of DNA tokens for a small sequence indicative of some feature.

The circuit will receive one token per clock cycle as input; the possible tokens are adenine (A), cytosine (C), guanine (G) and thymine (T). The circuit should, given the input sequence, set an output flag to 1 when the matching sequence ACT is found

somewhere in the input or 0 otherwise. You can assume the inputs are infinitely long, i.e., the circuit should just keep searching forever and set the flag when the match is a success.

a. Design a circuit to perform the required task, show all your working and explain any design decisions you make along the way.
b. Now imagine you are asked to build two new matching circuits which should detect the sequences *CAG* and *TTT* respectively. It is proposed that instead of having three separate circuits, they are combined into a single circuit that matches the input sequence against one matching sequence selected with an additional input. Describe **one** advantage and **one** disadvantage you can think of for the two implementation options.

10. NAND is a universal logic gate in the sense that the behaviour of NOT, AND and OR gates can be implemented using only NAND. Show how this is possible using a truth table to demonstrate your solution.

11. A revolutionary, ecologically sound washing machine is under development by your company.

When turned on, the machine starts in the *idle* state awaiting input. The washing cycle consists of the three stages: *fill* (when it fills with water), *wash* (when the wash occurs), *spin* (when spin dying occurs); the machine then returns to *idle* when it is finished. Two buttons control the machine: pressing B_0 starts the washing cycle, pressing B_1 cancels the washing cycle at any stage and returns the machine to *idle*; if both buttons are pressed at the same time, the machine continues as normal as if neither were pressed.

a. You are asked to design a logic circuit to control the washing machine behaviour. Draw a diagram that shows the washing machine states and the valid transitions between them.
b. Draw a state transition table for a state machine that could be used to implement your diagram.
c. Using the Karnaugh map technique, derive logic expressions which allow one to compute the next state from the current state in your table; note that because the washing machine is ecologically sound, minimising the number of logic gates is important.

Chapter 3
Hardware Design Using Verilog

If I designed a computer with 200 *chips, I tried to design it with* 150*. And then I would try to design it with* 100*. I just tried to find every trick I could in life to design things real tiny.*

– S. Wozniak

Abstract The goal of this chapter is not to give a comprehensive guide to Verilog development but to provide enough of an introduction that the reader can understand and implement concepts used as examples in subsequent sections. Although these concepts become increasingly complex they build on the basic principles outlined in this chapter and, as such, it should provide enough of a reference that no other text is necessary. We omit the study of low-level concepts like transistor-level design, subjects related to Verilog such as formal verification, and advanced topics such as the Programming Language Interface (PLI).

3.1 Introduction

3.1.1 The Problem of Design Complexity

Looking back as recently as fifty years ago, the number of components used to construct a given circuit was in the range of 10 to 100, all selected from a small set of choices. Operating at this scale, it was not unrealistic for a single engineer to design an entire circuit on paper and then sit with a soldering iron to construct it by hand. Malone describes one extreme example of what could be achieved [38, Pages 148–152]. Development of a floppy disk controller for the Apple II was, in early 1977, viewed as a vital task so that the computer could remain competitive in a blossoming market. Steve Wozniak, a brilliant technical mind and co-founder of Apple, developed and improved on existing designs and within two weeks had a working circuit. Not content with eclipsing the man-hour and financial effort required by other companies to reach the same point, Wozniak redesigned and optimised the entire circuit

over a twelve-hour period after spotting a potential bug before it was sent into production !

As circuit complexity has increased over the years, and lacking a Wozniak to deploy in every design company, this model fails to scale. For example, a modern computer processor will typically consist of many millions of transistors and related components. Just as designing robust, large-scale software is a challenge, designing a million-component circuit is exceptionally difficult. The sheer number of states the circuit can be in requires splitting it into modular sub-systems, the interface between which must be carefully managed. Without doing this, verifying the functional correctness of the circuit is next to impossible; one must instead verify and test each module in isolation before integrating it with the rest of the design. Physically building the circuit by hand is equally impossible because of the size of the features: one would need an exceptionally tiny soldering iron to wire connections together ! Furthermore, one would need almost infinite patience because of the number of connections. Since humans are bad at repetitive tasks like this, errors will inevitably be introduced. Coping with design constraints such as propagation delay becomes massively time consuming since the act of component layout and connection with so many items is a mind-bogglingly complex optimisation problem. The increased number of components introduces further problems; issues of power consumption and heat dissipation start to become real problems.

Several key advances have tamed the problems in this gloomy picture. Certainly improvements in implementation and fabrication technologies that allow miniaturisation of components have been an enabling factor, but the advent of **design automation** (i.e., software tools to help manage the complexity of hardware design) has arguably been just as important.

3.1.2 Design Automation as a Solution

Verilog is a **Hardware Description Language (HDL)**. A language like C is used to describe software programs that execute on some hardware device; a language like Verilog can describe the hardware device itself. Using a HDL, usually within some form of **Electronic Design Automation (EDA)** software suite, offers the engineer similar benefits to a programmer using a high-level language like C over a low-level machine language.

The Verilog language was first developed in 1984 at a company called Gateway Design Automation as a hardware modelling tool. Some of the language design goals were simplicity and familiarity; as a result Verilog resembles both C and Pascal with many similar constructs. Coupled with the provision of a familiar and flexible development environment, Verilog promotes many of the same advantages that software developers are used to. That is, the language makes it easy to abstract away from the implementation and concentrate on design; it makes design modularisation and reuse easier; associated tools such as simulators make test and verification easier and more efficient. There are two standards for the language: an initial version

termed Verilog-95 was improved and resubmitted to the IEEE for standardisation as Verilog-2001 or IEEE 1364-2001. Much of this improvement was driven by input from users in companies such as Motorola and National Semiconductor. This standardisation, along with the production of software to synthesise real hardware from Verilog descriptions at a variety of abstraction levels, has made the language a popular choice for engineers.

Verilog **models**, so-called because they are somewhat abstract models of physical circuits, can be described at several different levels. Each level is more abstract than the last, leaving less work for the engineer and allowing them to be more expressive:

Switch Level At the lowest level, Verilog allows one to describe a circuit in terms of transistors. Although this gives very fine-grained control over the circuit composition, developing large designs is still a significant challenge since the building blocks are so small.

Gate Level Above the transistor level, and clearly more attractive for actually getting anything useful done, is the concept of gate-level description. At this level, the designer describes the functionality of the circuit in terms of basic logic gates and the connections between them; this is somewhat equivalent to the level of detail in the previous chapter.

Register Transfer Level (RTL) RTL allows the designer to largely abstract away the concept of logic gates as an implementation technology and simply describe the flow of data around the circuit and the operations performed on it.

Behavioural Level The most abstract and hence most expressive form of Verilog is so-called behavioural-level design. At this level, the designer uses high-level constructs, similar to those in programming languages like C, to simply specify the required behaviour rather than how that behaviour is implemented.

Roughly speaking, development of real hardware from a Verilog model is a three-phase process which we describe as being managed by a suite of software tools called the Verilog **tool-chain**:

Simulate During the first phase, the Verilog model is fed into a software tool which simulates the behaviour of the circuit. This simulation is highly accurate, modelling how gates and signals change as time progresses, but is abstract in the sense that many constraints of real hardware are not considered. Since by using software simulation it is easy to alter the model and debug the result, a development cycle is relatively quick. As a result, the simulation phase is typically used to ensure a level of functional correctness before the model is processed into real hardware. That is, if the model simulates correctly, then one can progress to implementing it in real hardware with a greater confidence of success.

Synthesise The second phase of development is similar to the compilation step whereby a C program is translated into low-level machine language. The Verilog model is fed into synthesis software which converts the model into a list of real hardware components which implement the required behaviour. This is often called a **netlist**. Optimisation steps may be applied by the synthesiser to improve the time or space characteristics of the design. Such optimisations are usually

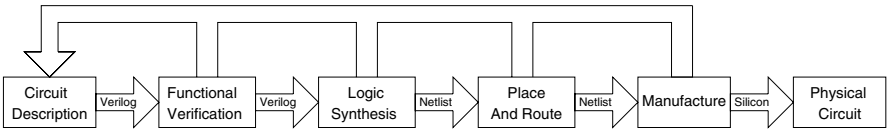


Figure 3.1 A waterfall style hardware development cycle using Verilog.

conceptually simple but repetitive: ideal fodder for a program but not so for the engineer.

Implement The synthesis phase produces a list of components and a description of how they are connected together to form the implementation. However, a further step is required to translate this description into real hardware. Firstly, we need to select components from the implementation technology with which to implement the components in the netlist. For example, on an FPGA it might be advantageous to implement an AND gate one way whereas on an ASIC a different method might be appropriate. Secondly, we need to work out where the components should be placed and how the wiring between them should be organised or routed. Long wires are undesirable since they increase propagation delay; the place and route process attempts to minimise this through intelligent organisation of the components. Again, this is a problem which is well suited to solution by a computer but not by the engineer.

In reality, these steps form part of a waterfall style development cycle (or similar) which iterates each stage to remove errors after some form of verification.

3.2 Structural Design

3.2.1 Modules

Programs written in C can be thought of as collections of functions that use each other to perform some computation. The computation is determined by how the functions call each other and by what goes on inside each function. The analogous construct in Verilog is a **module**. Basically speaking, a module defines the **interface** of a hardware component, i.e., the inputs and outputs, and how the component does computation to produce the outputs from the inputs and any stored state.

Like functions, modules provide us with a first layer of abstraction: we can view them as a black-box, with inputs and outputs, without worrying how the internals work. For example, the multiplexer device introduced previously selects between several values to produce a result; we might draw such a device as previously shown in Figure 2.16a. From a functional point of view, we do not really care what is inside

```
module mux2_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire s0 );

  ...

endmodule
```

Listing 3.1 The interface for a 1-bit, 2-way multiplexer.

```
module mux2_1bit( r, i0, i1, s0 );

  output wire  r;
  input  wire i0;
  input  wire i1;
  input  wire s0;

  ...

endmodule
```

Listing 3.2 The interface for a 1-bit, 2-way multiplexer.

the component. That is, in order to use a multiplexer we do not care how it works or what gates are used within it, just what it does.

We can describe the interface of Figure 2.16a, i.e., the inputs and outputs rather than the internals, using the Verilog module definition shown in Listing 3.1. A module definition like this specifies a name or **identifier**, this one is called `mux2_1bit`, and a list of inputs and outputs which it uses for communication with other modules. We term the inputs and outputs defined by a module as the **port list**; each input or output is a **port** over which it communicates, the direction of communication is determined by the `input` and `output` keywords. In this case, we have replaced the **body** of the module, which determines how it works, with some continuation dots that we will fill in later.

Listing 3.2 offers a different way of defining the same interface. This time, instead of describing the input and output characteristics of each port inline with the definition, we simply name the ports and give them a type inside the module. Although the two methods are largely equivalent, there are some advantages to the second method which we will see later. Typically, for simple modules the first method is more compact and so is preferred unless there is a good reason not to use it.

3.2.2 Wires

In Verilog, a **net** is the term used to describe the connection between two modules. By net we essentially we mean a **wire** which is the exact analogy of the same thing in real life: a conducting medium that allows a signal to be sent between the two

```
1 wire              w0;      // 1-bit unsigned wire
2 wire        [3:0] w1;      // 4-bit unsigned wire vector
3 wire        [0:3] w2;      // 4-bit unsigned wire vector
4 wire        [4:1] w3;      // 4-bit unsigned wire vector
5 wire signed [3:0] w4;      // 4-bit signed wire vector
```

Listing 3.3 Several different examples of wire definition.

ends. We say that we **drive** a value onto one end before it propagates along the wire and can be read at the other end. We define Verilog wires using a similar syntax to variable declaration in C, as shown in Listing 3.3.

Line 1 defines a wire called `w0` capable of carrying 1-bit values. Although 1-bit values are fine for some situations, often we will want to communicate larger values. So-called **wire vectors** allow us to do exactly this by bundling 1-bit wires into larger groups. Since we can select the individual wires from a given wire vector, we can think of the vector as either as an array of four 1-bit wires, or a single compound wire that can carry 4-bit values. Line 2 demonstrates this by defining a wire vector called `w1` which can carry 4-bit values. One can read the range notation `[3:0]` as 3 down to 0 or 3,2,1,0. This is a means of specifying the size of the wire vector, as well as the order and index of wires within it. For example, Line 3 defines another 4-bit wire vector but the order of the range is reversed, the left-most bit of `w2` is bit 0 while the right-most is bit 3. In the previous example, the left-most bit was bit 3 while the right-most was bit 0. Line 4 uses a third variation where the order of wires within the wire vector is the same as `w1` but their index is changed. This time the left-most bit of `w3` is numbered 4, rather than 3, while the right-most is numbered 1, rather than 0. These variations might seem confusing but they are useful in practise. For example, reversing the order of the bits in a wire allows us to easily change the endianness and hence easily comply with demands set by the system we want to connect our design to.

It is worth noting that more recent Verilog language specifications include the facility to declare signed wire vectors, i.e., wire vectors that carry signed, twos-complement values rather than unsigned values. For example, Line 5 in Listing 3.3 declares a 4-bit signed wire vector called `w4`; it can take the values between -8 and 7 inclusive. This differs from Line 2 where `w1` is declared as a 4-bit unsigned wire vector capable of taking the values between 0 and 15 inclusive. Clearly this is a matter of interpretation to some extent since one can view a given bit pattern as representing a signed or unsigned value regardless of the wire vector type. However, the addition of the signed keyword "hints" to the Verilog tool-chain that, for example, if the value on signed wire `w4` is inspected or displayed somehow it should be viewed as signed.

Value	Meaning
`1'b0`	Logical low or false
`1'b1`	Logical high or true
`1'bX`	Undefined or unknown value
`1'bZ`	High impedance value

Table 3.1 A list of the four different value types.

Strength	Meaning
`supply`	Strongest
`strong`	
`pull`	
`large`	
`weak`	↓
`medium`	
`small`	
`highz`	Weakest

Table 3.2 A list of the eight different strength types.

3.2.3 Values and Constants

In a perfect hardware system, a wire should only ever take the values 0 or 1 (i.e., true and false) written in Verilog as `1'b0` or `1'b1`. However, physical properties of such systems dictate they are never perfect. As a result, Verilog supports four different values, shown in Table 3.1, to model the reality of imperfect hardware. In a sense, there is a similar motivation here as in the discussion of 3-state logic in Chapter 2.

The **undefined** or **unknown** value `1'bX` is required to model what happens when some form of conflict occurs. It is not that we do not care what the value is, we genuinely do not know what the value is. As a simple example, consider driving the value `1'b1` onto one end of a wire and the value `1'b0` onto the other end, what value does the wire take ? The answer is generally undefined, but roughly depends on how strong the two signals are. Verilog offers a way to model how strong a value of `1'b0` or `1'b1` is; one simply prefixes the value with one of the strength types shown in Table 3.2. However, this sort of assumption should be used with care: generally speaking, if you rely on an unknown value being resolved to something known, there is probably a better way to describe the design and avoid the problem. The **high impedance** value `1'bZ` is a kind of pass-through described previously in the context of 3-state logic. If high impedance collides with any value, it instantly resolves to that value. This is useful for modelling components which must produce

some continuous output but might not want to interfere with another component using the same wire.

Constant values, which one can use to drive wires or to compare against values read from wires for example, can be written in a variety of ways using the general format

```
[<sign>] [<size>'<base>] <value>
```

such that the fields in square brackets are optional. The `<sign>` field is an optional minus symbol to denote negative constants; `<size>` denotes the number of bits in the constant; `<base>` is the number base the constant is expressed in, for example `H` or `h` for hexadecimal, `D` or `d` for decimal and `B` or `b` for binary; and value is the actual constant. So, for example we have that:

- `2'b10` is a 2-bit binary value $10_{(2)}$ or $2_{(10)}$ or $2_{(16)}$.
- `4'hF` is a 4-bit hexadecimal value $1111_{(2)}$, $15_{(10)}$ or $F_{(16)}$.
- `8'd17` is an 8-bit decimal value $17_{(10)}$ or $00010001_{(2)}$ or $11_{(16)}$.
- `3'bXXX` is a 3-bit unknown binary value.
- `8'hzz` is an 8-bit high impedance value.
- `4'b10XZ` is a 4-bit binary value where bit zero is the high impedance value, bit one is the unknown value, bit two is logical low and bit one is logical high. This value does not really translate into another base; although some of the bits are defined, because some are not the whole value is undefined.

Where size is not important, one can omit the `<size>` field and accept a default. Likewise, one can omit the `<base>` field to accept the default of decimal numbers so that the constant `10` is the same as `4'd10`. Clearly we can also specify signed constants by prefixing them with a negation symbol such as `-10` or `-4'd10`.

3.2.4 Comments

Like most other programming languages, Verilog allows one to **comment** source code. Comments are used with the same syntax as C and Java so that single-line comments

```
// comment
```

and multi-line comments

```
/*
comment
comment
...
*/
```

are both possible. It goes without saying that since Verilog models can contain just as many subtle implementation issues as a C program, comments are a vital way to convey meaning which might not be present in the source code.

```
1 module or_gate( output wire r,
2                 input  wire x,
3                 input  wire y );
4
5   ...
6
7 endmodule
```

Listing 3.4 The interface for a 2-input OR gate.

```
1 or t0( r, x, y       ); // 2-input or gate instance
2 or t1( r, x, y, z    ); // 3-input or gate instance
3 or t2( r, w, x, y, z ); // 4-input or gate instance
```

Listing 3.5 Instantiating primitive gates with multiple inputs.

3.2.5 Basic Instantiation

Continuing the analogy between C functions and Verilog modules, a function clearly does not do anything until it is called: the function is just a description. Equally, a module does not do anything until it is **instantiated**. Instantiating a module is the equivalent of actually placing a physical copy of it down on our design. If we instantiate a multiplexer module, this is the analogy of taking a multiplexer component from a box and placing it on a circuit board.

Verilog offers a number of **primitive modules** that represent logic gates. In a sense you can think of primitive modules as similar to the C standard library which makes basic operations available to *all* C programs. We can connect these together to describe more complex circuits and as such they offer a convenient way to introduce the instantiation syntax. Consider a primitive two input OR gate. Although it is provided for us by Verilog, we can think of it having an interface similar to that shown in Listing 3.4 (we deliberately name it `or_gate` and not `or` to avoid conflict with the built-in name). In fact, Verilog is nicer to us than simply providing a 2-way version, it provides multi-input versions of these modules; one simply adds more inputs to the instantiation as shown in Listing 3.5. For example, the instantiation of `t1` in Line 2 names three inputs `x`, `y` and `z`.

Using this primitive OR gate module, and similar ones for AND, NOT and XOR, we can start filling in the module body of Listing 3.1 so it matches the multiplexer design from Figure 2.16a whose truth table is shown in Table 2.10a. Listing 3.6 describes an implementation by first defining some internal wires to connect the components and then instantiating primitive modules to represent those components. Each instantiation is built from a few syntactic parts: the type of the module, an identifier for the instance, and a port list. In this case, the types are `not`, `and` and `or`; the identifiers are `t0`, `t1`, `t2` and `t3`. The port list describes how wires connect to the communication ports of the module. For example, in the case of the `or` gate we have connected wires `w1` and `w2` to the `or` gate input ports, and output of the multiplexer `r` to the `or` gate output port. As another example, consider the imple-

```
module mux2_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire s0 );

  wire w0, w1, w2;

  not t0( w0, s0     );

  and t1( w1, i0, s0 );
  and t2( w2, i1, w0 );

   or t3( r,  w1, w2 );

endmodule
```

Listing 3.6 An implementation of a 1-bit, 2-way multiplexer using primitive modules.

```
module add_1bit( output wire co,
                 output wire  s,
                 input  wire ci,
                 input  wire  x,
                 input  wire  y );

  wire w0, w1, w2;

  xor t0( w0, x,  y  );
  and t1( w1, x,  y  );

  xor t2(  s, w0, ci );
  and t3( w2, w0, ci );

   or t4( co, w1, w2 );

endmodule
```

Listing 3.7 An implementation of a full-adder using primitive modules.

mentation of a full-adder in Listing 3.7. Exactly the same principles apply as in the multiplexer implementation: we again wire together a number of primitive gate instances to replicate the functionality required by the truth table shown in Table 2.8b.

Stated in the above form, module instantiation appears very similar to a C function call which will also typically have an identifier and arguments. But there are some crucial differences between the two:

- Typically, a compiler translates each function declared in a C program into *one* implementation in the executable. That is, two separate calls to the same function actually use the same code: they call the same function implementation. With module instantiation, rather than being shared, *each* instance is a *separate* item of hardware.
- Within a typical C program, each function call in the program is executed *sequentially* so that at any given time we know where in the program execution is: it cannot be in two functions at the same time. With a Verilog design, every mod-

```
module mux4_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire i2,
                  input  wire i3,
                  input  wire s0,
                  input  wire s1 );

  wire w0, w1;

  mux2_1bit t0( w0, i0, i1, s0 );
  mux2_1bit t1( w1, i2, i3, s0 );
  mux2_1bit t2( r,  w0, w1, s1 );

endmodule
```

Listing 3.8 An implementation of a 1-bit, 4-way multiplexer using user-defined modules.

ule instance executes in *parallel* with every other instance so that one module can be doing things at the exact same time as another.

Since each instance executes in parallel with every other, the `or` gate used above can be described as **continuous** in the sense that it does not wait for the inputs to be ready before computing the output: it is always computing an output. We could just as well write Listing 3.6 with the `or` gate instantiated before both the `and` gates. In C, this would make no sense because now the results `w1` and `w2` are not produced before they are used. In Verilog however, all three modules are continuously working in parallel so it does not matter what order we instantiate them in as long as they are wired together correctly.

Clearly this continuous behaviour can cause problems if you are used to the sequential nature of C. For example, what happens if some of the inputs have values driven onto them while others do not ? Since the module is continually generating an output, the output must be something, but it will typically be resolved as the undefined value. When executing a C program, the processor deals with this issue for us by forcing each instruction to execute in order; from a functional perspective a given instruction never starts until previous instructions are completed. In Verilog however, we are left on our own to enforce such rules.

This presents us with the first challenge of timing within Verilog models: we need to be sure that when we use the output of a module, enough time has elapsed to generate it. We have already seen that propagation delay in gates takes some time which could impact on when the output becomes valid. As the model size increases these small delays can accumulate and present a significant delay which needs to be accommodated. In the simple combinatorial circuits we have looked at thus far, timing is not a major problem since we typically just drive the input values and wait until they generate the output. With models that store state, which we encounter in Section 3.4, timing becomes much more important. For example, we need to ensure that we only set the value of our state with values that are valid, i.e., are not undefined.

3.2.6 Nested Instantiation

Primitive modules are just like modules you describe yourself except that they are provided by Verilog. Therefore we can consider a design composed of several of our own modules in the same way as we did above with the 2-way multiplexer. Consider, for example, extending the 2-way multiplexer so it has four inputs selected between using two control signals. One can think of at least two methods of implementing this design which were described previously: either we use the same primitive gates as in the 2-way case or we build it using our previously defined module. Ignoring which is the most efficient in terms of the number of gates, the latter method clearly leans on one of the advantages of Verilog in that reuse of modules is made very easy. Listing 3.8 takes advantage of this by implementing a 4-way multiplexer called `mux4_1bit` using three instances of the 2-way multiplexer `mux2_1bit`. Exactly the same rules apply when instantiating `mux2_1bit` as when instantiating a primitive `or` gate: we have a type, an identifier and a port list for each instance and each instance of our 2-way multiplexer operates continuously and in parallel with every other.

Verilog instantiation creates a **hierarchy** of instances. Each instance is said to contain those modules it in turn instantiates. To keep track of this hierarchy, Verilog outlines a simple naming scheme. For example, the 4-way multiplexer in Listing 3.8 has three instances of a 2-way multiplexer named `t0`, `t1` and `t2`. Within each of these 2-way multiplexers there is a `not` gate named `t0`, two `and` gates named `t1` and `t2` and an `or` gate named `t3`. Hierarchical naming means each primitive gate has a unique name: the `not` gate within 2-way multiplexer `t1` is called `t1.t0` while the one in multiplexer `t2` is called `t2.t0`. This is a useful property since we can embed debugging mechanisms within a module and determine exactly where the messages are coming from; without the naming scheme, all debugging from the module `mux2_1bit` would be mixed together and become useless with respect to locating errors.

3.2.7 User-Defined Primitives

For some simple functions, building a module from gates is overly verbose. Additionally, it defeats one of our reasons to use Verilog which was to hand off as much work to the Verilog tool-chain as we can. The **User-Defined Primitive (UDP)** offers a more convenient way to build simple modules by essentially allowing the designer to specify the truth table and leave the implementation to the tool-chain. Given such a truth table, the tool-chain tries to find the most optimal way to realise it, using for example the Quine-McCluskey method.

UDPs have a similar interface to modules. For example, we might define our 2-way multiplexer as shown in Listing 3.9. The body of the UDP is the truth table which states, given some combination of input, what output should be produced. Each line in the table lists the values of the input ports followed by a colon and

```
primitive mux2_1bit( output  r,
                     input  i0,
                     input  i1,
                     input  s0 );
  table
    0 ? 0 : 0;
    1 ? 0 : 1;
    ? 0 1 : 0;
    ? 1 1 : 1;

    ? ? X : X;
  endtable

endprimitive
```

Listing 3.9 An implementation of a 1-bit, 2-way multiplexer using a UDP.

then the value required on the output port. We are free to include the unknown value `1'bX` but the high impedance value `1'bZ` is treated as `1'bX`. We are also free to include don't care states in the table as we might with a Karnaugh map; such states are written using a question mark. Thus, the first line above can be read as "when both `i0` and `s0` are 0 and `i1` is any value, set the result `r` to 0". The last line can be read as "when `s0` is unknown and both `i0` and `i1` are any value, set the result `r` to unknown".

Compared to the description of the same module in terms of gates, this seems a much nicer way to do things. However, there are some caveats to consider when defining a UDP:

- There can only be one output port although there can be as many inputs as required; the output port must be specified first in the port list.
- Both input and output ports can only be single wires; wire vectors are not allowed.

3.3 Higher-level Constructs

Expressing a design purely in terms of basic logic gates is analogous to writing software directly in a low-level machine language: it offers great control over the result but progress requires significant effort on the part of the programmer. To combat this, we can use RTL style design to construct Verilog models at a higher-level of abstraction. Rather than worrying exactly how the functionality is realised, RTL allows us to concentrate on how data flows around the design and the operations performed on it. Clearly there is some overlap between RTL and gate-level design; essentially RTL offers a short hand way to describe combinatorial logic.

```
module nand_gate( output r,
                  input  w,
                  input  x,
                  input  y,
                  input  z );

  assign r = ~( w & x & y & z );

endmodule
```

Listing 3.10 Using a continuous assignment to replicate the behaviour of a primitive module.

```
module mux2_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire s0 );

  assign r = ( s0 ? i1 : i0 );

endmodule
```

Listing 3.11 An implementation of a 1-bit, 2-way multiplexer using a continuous assignment.

3.3.1 Continuous Assignments

In Section 3.2 we described the operation of primitive gates as continuous in the sense that they were continuously calculating outputs from their inputs rather than waiting for the inputs to be available. We can use a higher-level, **continuous assignment** RTL statement to do the same thing. Such a statement looks similar to an assignment in C in the sense that it has left-hand side (LHS) and right-hand side (RHS) expressions. The LHS is said to be assigned a value calculated by evaluating the RHS. In a C program we typically expect the LHS to be a variable; in a Verilog continuous assignment the LHS is a wire or wire vector. Furthermore, and unlike in a C program, a Verilog continuous assignment is continually being executed rather than only being executed when control-flow reaches the assignment. This is a fundamental difference and relates to the fact that instantiations of Verilog modules, which a continuous assignment will be translated into by the tool-chain, are continuously operating in parallel with each other.

The beauty of the construct is that one can use a huge range of C style logical and arithmetic operators as part of the RHS expression; Table 3.3 contains a list of them, their types and number of operands. Consider the construction of a 4-input NAND gate shown in Listing 3.10 (again we name the module `nand_gate` and not `nand` to avoid conflict with the built-in name). Of course, one could use a built-in primitive module to achieve the same thing, but this simple assignment neatly demonstrates the expressive power of a continuous assignment. Instead of several module instantiations and wire connections, we simply write the required RHS expression and let the Verilog tool-chain convert this into an efficient low-level implementation. Virtually all C operators are permitted so, for example, we can re-

Type	Symbol	Meaning	Operands
Arithmetic	*	multiply	2
	/	divide	2
	+	add	2
	-	subtract	2
	%	modular reduction	2
	**	exponentiation	2
Logical	!	logical NOT	1
	&&	logical AND	2
	||	logical OR	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
	==	equal to	2
	!=	not equal to	2
	===	case equal to	2
	!==	case not equal to	2
Bitwise	~	bitwise NOT	1
	&	bitwise AND	2
	|	bitwise OR	2
	^	bitwise XOR	2
Reduction	&	reduce AND	1
	~&	reduce NAND	1
	|	reduce OR	1
	~|	reduce NOR	1
	^	reduce XOR	1
	~^	reduce XNOR	1
Shift	<<	logical left-shift	2
	<<<	arithmetic left-shift	2
	>>	logical right-shift	2
	>>>	arithmetic right-shift	2
Misc	[...]	selection	*n*
	{ ... }	concatenation	*n*
	{ { ... } ... }	replication	*n*
	... ? ... : ...	conditional	3

Table 3.3 A list of Verilog operator types.

x	$x \wedge y$
0	1
1	0
X	X

(a) NOT.

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1
0	X	0
1	X	X
X	0	0
X	1	X
X	X	X

(b) AND.

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1
0	X	X
1	X	1
X	0	X
X	1	1
X	X	X

(c) OR.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0
0	X	X
1	X	X
X	0	X
X	1	X
X	X	X

(d) XOR.

Table 3.4 Logical operators in the presence of unknown values.

implement our 2-way multiplexer using an inline-conditional expression as shown in Listing 3.11. In this case, the LHS is the output wire `r` whose value is calculated by evaluating the RHS expression. If wire `s0` is 1, the expression evaluates the value of `i1`, otherwise it evaluates to the value of `i0`.

One subtle caveat exists with the analogy between gates and logical operators: the latter deals explicitly with cases where one or more of the inputs are the unknown or high impedance values `1'bX` or `1'bZ`. Firstly, in this context the value `1'bZ` is treated as unknown or `1'bX`. Then, the rules in Table 3.4 are applied to describe the output based on the inputs. Clearly there are additional entries versus traditional truth tables for NOT, AND, OR and XOR. These additional entries show that, for example, if one were to AND together two values using `1'b0 & 1'bX`, then the result would be `1'b0`. Intuitively this might seem odd: how can one decide on a result if one of the inputs is unknown ? In reality, the properties of AND mean this is possible since when the first input is 0, it does not matter what the other input is because the output will always be 0. Similar shortcuts exist for the other operators. For example, if either input of an OR operator is 1 it does not matter what the other one is (even if it is unknown) because the output will always be 1.

3.3.2 Selection and Concatenation

Although the RHS expression of a continuous assignment closely resembles a C expression, the fact that C is designed for writing software programs means operations we expect in hardware are absent. To address this, Verilog includes **selection**, **concatenation** and **replication** operators which split, join and replicate wires and wire vectors.

```
module add_1bit( output wire co,
                 output wire  s,
                 input  wire ci,
                 input  wire  x,
                 input  wire  y );

  assign { co, s } = x + y + ci;

endmodule
```

Listing 3.12 An implementation of a full-adder using RTL-level design features.

Given the definitions in Listing 3.3, we can split the wire vector `w1` into parts using an operator similar to array sub-scripting in C. The expression

```
w1[0]
```

selects wire number zero from the vector `w1`; the type of this expression is a 1-bit wire. If `w1` takes the value `4'b1010`, then the expression `w1[0]` evaluates to `1'b0`. We can extend this selection to include ranges so that the expression

```
w1[2:0]
```

is a 3-bit wire we have split from `w1` by selecting bits two, one, and zero. The expression evaluates to `3'b010`.

In a similar way, we can join together wires using the concatenation operator so that the expression

```
{ w0, w1 }
```

is a 5-bit wire vector. Wire number four in this vector is equal to the value of `w0` while wires three, two, one and zero are equal to the corresponding wires in `w1`. So if `w0` takes the value `1'b0` then the above expression evaluates to `5'b01010`. Note that there is something of a duality between selection and concatenation since the expression

```
w1[2:0]
```

is in fact equivalent to

```
{ w1[2], w1[1], w1[0] }
```

where in both cases we are creating a 3-bit wire vector with component wires drawn from the vector `w1`. Finally, the replication operator allows us to create repeated sequences by specifying a replication factor and a value. The expression

```
{ 4{w1[0]} }
```

is equivalent to the expression

```
{ w1[0], w1[0], w1[0], w1[0] }
```

with both evaluating to `4'b0000`.

As a demonstration of these operators in a practical context, consider a simplification of the previous gate-level full-adder design. The power of using a more

abstract description is shown in Listing 3.12. Instead of a number of very primitive module instances connected together, we use a single, expressive continuous assignment. The RHS of the assignment is easy to understand: it simply adds together the two inputs `x` and `y` and the carry-in `ci`. The LHS uses a more cryptic concatenation of two wires `co` and `s`, the result is a 2-bit wire vector. Since the result of the RHS, i.e., adding together the three input wires, is a 2-bit result. This forces the top bit into the carry-out and the bottom bit into the sum: exactly what we want. By being more expressive we not only make the module easier to read, we also open the opportunity for the Verilog tool-chain to optimise the design for us. For example, if the implementation technology offers a particularly efficient hardware structure for an adder, the tool-chain can spot the operator in our code and capitalise on this fact. In this case the advantages are perhaps marginal, but for more complex designs it can make a real difference to the end result.

3.3.3 Reduction

The second class of operation which will be alien to C programmers are those of **reduction**. Each Verilog reduction operator takes one operand and produces a 1-bit result; essentially it applies a logical operator to all the bits in the operand, bit-by-bit from right to left. Again considering the wire definitions in Listing 3.3 and assuming `w1` is set to the value `4'b1010`, the reduction expression

```
|w1
```

is exactly equivalent to the expression

```
w1[0] | w1[1] | w1[2] | w1[3]
```

which evaluates to `1'b1`. Another example could be

```
&w1
```

which is exactly equivalent to the expression

```
w1[0] & w1[1] & w1[2] & w1[3]
```

and evaluates to `1'b0`. This sort of operation does not exist in C because it is word-oriented with the programmer seldom combining bits from the same word. In Verilog however, we work more freely and can easily extract and utilise bits from any value since they are always accessible as 1-bit wires.

3.3.4 Timing and Delays

To model issues of **timing** in a model, Verilog offers several ways to specify **delay**. In the context of continuous assignment, one can utilise **regular delay** to specify the time between changes to the RHS expression and the assignment to the LHS. One

```
reg               r0;       // 1-bit unsigned register
reg        [3:0] r1;       // 4-bit unsigned register vector
reg        [0:3] r2;       // 4-bit unsigned register vector
reg        [4:1] r3;       // 4-bit unsigned register vector
reg signed [3:0] r4;       // 4-bit signed register vector
reg               r5[7:0]; // 1-bit, 8-entry register array
reg        [3:0] r6[7:0]; // 4-bit, 8-entry register vector array
```

Listing 3.13 Several different examples of register definition.

can think of this as the analogy of propagation delay whereby it takes some length of time for values to propagate through the circuit representing the RHS expression before they result in a value on the LHS. For example, the assignment

```
assign #10 r = ( x | y );
```

uses a regular delay of ten time units; the value of `r` is assigned ten time units after any change to `x` or `y`. One caveat to this simple rule is that the change in `x` and `y` must persist for at least as long as the time delay for `r` to actually see the result. If `x` changes from 1 to 0 and then back again in less than ten time units, `r` will not change to represent this change.

3.4 State and Clocked Design

So far we have only considered combinatorial circuits which hold no state but simply compute some outputs from some input values. This is partly because wires cannot hold state, their value being maintained only as long as a signal is driven onto the wire. However, we will often want to maintain some state in a design so that computation is based not only on the input values but also on previous computation.

3.4.1 Registers

To maintain state, Verilog allows the definition of **registers** that retain their value even when not being continuously driven with that value; registers are more or less analogous to variables in a C program. While wires in Verilog are literally wires in real hardware, registers are implemented using flip-flops so that the stored value is retained as long as the flip-flop is powered. While a wire is continuously updated with whatever value drives it, a register is only updated when a new value is assigned into it.

Definition of registers is essentially the same as definition of wires, one simply replaces the `wire` keyword with `reg`; Listing 3.13 describes some examples. However, the introduction of registers adds some complication in terms of how one uses

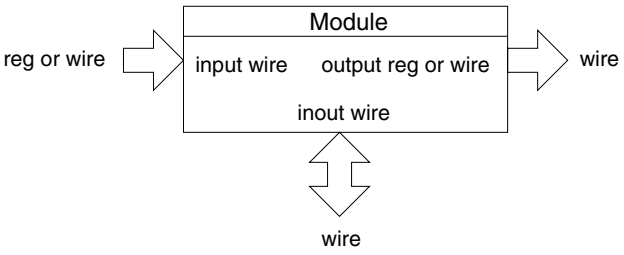


Figure 3.2 Connection rules for registers and wires.

inputs and outputs to and from modules. In short, only certain types can be used in each circumstance; Figure 3.2 describes these rules. For example, if some register `r` is defined within a module `x`, it can be named in the input port list to some module instance `y` within `x`. However, within `y` the input must be viewed as a wire: we cannot be sure that the input will always be a register (and hence retain state) so we must be pessimistic about its behaviour. In this description, we have introduced a new type of port, marked with the `inout` keyword, which can act as both an input and output.

Although one might describe a register vector above as like being an array of registers, this is not a great analogy since an array is usually a connection of separate values whereas a register vector typically represents one value. Verilog does offer "real" arrays as well, which are somewhat confusingly termed memories. The range of indices which address the memory is declared after the memory identifier as described in Line 6 and Line 7 in Listing 3.13. In these declarations we declare two memories: `r5` has eight entries numbered $7,6,\ldots,0$, each entry is a 1-bit register; `r6` also has eight entries indexed in the same way but each entry in the memory is a 4-bit register vector.

There are some constraints on where and when one can reference elements in a memory, but roughly speaking the syntax follows that of register vectors so that the expression

```
r5[2]
```

selects element 2, a 1-bit register, from memory `r5` while the expression

```
r6[1]
```

selects element 1, a 4-bit register, from memory `r6`. We can combine the two forms to select individual wires from this. The expression

```
r6[1][1:0]
```

selects element 1 from `r6` in the same way as above, then extracts bits 1 and 0 from the result to finally form a 2-bit vector.

```
begin
  x = 4'b0000;
  y = 4'b1111;
end
```

(a) Basic sequential block.

```
begin:name
  x = 4'b0000;
  y = 4'b1111;
end
```

(b) Named sequential block.

```
fork
  x = 4'b0000;
  y = 4'b1111;
join
```

(c) Basic parallel block.

```
fork:name
  x = 4'b0000;
  y = 4'b1111;
join
```

(d) Named parallel block.

Table 3.5 Examples of different types of behavioural code block.

3.4.2 Processes and Triggers

Blocks of statements are marked before and after using keywords that act in a similar way to the { and } braces in C programs. The `begin` and `end` keywords denote a **sequential block** of statements; each statement in the block is executed in the order they are listed, any timing control for a statement is relative to the previous statement. The `fork` and `join` keywords denote a **parallel block** of statements; each statement in the block is executed at the same time as the others, each timing control is relative to the start of the block. We can attach a hierarchical name to each block of code by appending a colon and identifier to the start of block keyword. Examples of each of these block definitions can be found in Table 3.5. Note that in a similar way to how { and } braces work in C programs, we can omit the start and end of block keywords for one line blocks.

All blocks must exist within a Verilog **process**. A process can be thought of as a parallel thread of execution. To rationalise this, consider using an RTL-style approach to drive values onto two wires via two continuous assignments. These assignments are always being evaluated; they are executing at the same time. How would one cope with this using a behavioural approach ? The `fork` and `join` type blocks could come in handy but more generally we should split our behavioural statements into two separate groups, or processes, that each deal with updating one of the values. The processes will execute in parallel with each other just like the continuous assignments, they simply use behavioural code rather than RTL-style constructs to implement the required functionality.

There are two types of Verilog process, one marked with the `always` keyword and one with the `initial` keyword. An `always` process is like an infinite loop in that execution of the content starts at the top, continues through the body and then restarts again at the top. In contrast, an `initial` process only executes once. Table 3.6a demonstrates the definition of an `always` process; we simply decorate

```
always
begin
  ...
end
```

(a) Untriggered `always` processes are possible but poor design practice.

```
always @ ( x )
begin
  ...
end
```

(b) Triggering by value change.

```
always @ ( posedge x )
begin
  ...
end
```

(c) Triggering by positive edge.

```
always @ ( negedge x )
begin
  ...
end
```

(d) Triggering by negative edge.

Table 3.6 Examples of different process definitions and triggers.

a normal block with the `always` keyword. Basic processes are somewhat simple in that they either execute just once or loop forever; there is no way to specify when they should be executed. We can add a **trigger**, or **sensitivity list**, to a process to specify this sort of information. Having looked at how digital logic works, examples are easy to come by: we might want a block of code to be triggered by a positive or negative signal edge or level. Table 3.6b-d demonstrate the addition of triggers which cause the associated block to execute when the value of `x` changes from anything to anything; when a positive edge in `x` is encountered, that is a change in value from 0 to 1; and when a negative edge in `x` is encountered, that is a change in value from 1 to 0. Here we have included just one signal `x` in the sensitivity list, more can be added by separating the extra signals with commas; in this case we might specify, for example, that a process is triggered when either `x` *or* `y` changes value.

Note that Table 3.6a which represents an untriggered `always` block is considered bad design practice: without some form of delay, we are saying that every iteration of the block happens at the same time. Since this is clearly impossible, we need to ensure that there is some delay within the process or, more preferably, change the design so it is clear when each iteration of the process is triggered.

3.4.3 Procedural Assignments

Within a block of code, we can place a sequence of statements which are executed when the encompassing process is triggered. The most simple example of a state-

```
begin
  x  = 10;
  y  = 20 + x;
end
```

(a) Blocking procedural assignments.

```
begin
  x <= 10;
  y <= 20 + x;
end
```

(b) Non-blocking procedural assignments.

Table 3.7 Examples of different procedural assignments.

ment is the **procedural assignment** which evaluates a RHS expression and assigns the value to an LHS expression. Such assignments come in two flavours, **blocking assignments** and **non-blocking assignments**. A blocking assignment is said to block subsequent statements in the sense that they only execute after it has completed. In contrast, a non-blocking assignments allow subsequent statements to execute at the same time as it. For example, consider the two blocks in Table 3.7. The left-hand example uses blocking assignments; the assignment to `y` only executes once the assignment to `x` is finished. This means that when the assignment to `y` executes, `x` will definitely have been assigned the value $10_{(10)}$. However, the right-hand process uses non-blocking assignments; the assignments to `x` and `y` execute at the same time. Here, this causes a problem because we can no longer guarantee that `x` will definitely have been assigned the value $10_{(10)}$ before the assignment to `y` is executed; this means `y` potentially gets the wrong result. Problems like this are avoided by considering a data-flow graph and avoiding dependent non-blocking assignments. Additionally, we should not mix blocking and non-blocking assignments within a given block of code so that it is clear when a particular assignment will occur.

Procedural assignments can incorporate timing and delay specifiers in the same way as continuous assignments. However, it is important to again note the difference between the two: continuous assignments are continually being executed while procedural assignments are only executed when their process block is triggered. Additionally, note that procedural assignments can only use registers as the LHS while continuous assignments can only use wires. This makes sense since the execution of a procedural assignment is time dependent and hence requires the LHS to retain state.

Either way, we are now in a position to actually design a meaningful module using behavioural-level Verilog. Recall that a D-type flip-flop is a component that can store 1-bit values; it has an output Q which represents the currently stored value an input D which is used to set the value when a positive edge is driven onto the enable signal *en*. Listing 3.14 replicates this interface and implements the required behaviour using two processes. Firstly, we define a 1-bit register called `t` which is used to represent the value stored within the flip-flop; we use a continuous assignment to set the flip-flop output `Q` to match the value `t`. Using an `initial` process, we ensure that `t` is set to 0 to start with. Then, we use an `always` process which is triggered whenever a positive edge is detected on the input `en`. Whenever this

```
module dtype_ff( input  wire  D,
                 output wire  Q,
                 input  wire en );
  reg t;

  assign Q = t;

  initial
  begin
    t = 0;
  end

  always @ ( posedge en )
  begin
    t = D;
  end

endmodule
```

Listing 3.14 An implementation of a D-type flip-flop using behavioural Verilog.

occurs the code within is executed; this means the value of input `D` is assigned to our register `t` and Q is updated in turn by the continuous assignment.

3.4.4 Timing and Delays

We previously saw how one can specify regular delays on continuous assignments to model propagation delay. This is also true of procedural assignment, although the semantics are slightly different. Using the following

```
#10 r = ( x | y );
```

simply specifies that the assignment should be executed 10 time units after the previous statement completed. This essentially defers execution of the entire assignment statement for some period of time. Trying to work out how timing delays interact with non-blocking forms of assignment is difficult and generally it is better not to mix the two forms.

In contrast to regular delays, **intra-assignment delay** simply defers update of the LHS. For example, by moving the timing specifier to the RHS, the assignment

```
r = #10 ( x | y );
```

executes immediately after the previous statement completes. The RHS is evaluated straight away but the assignment of the result to the LHS is only performed 10 time units later.

```
module dtype_ff( input  wire   D,
                 output wire   Q,
                 input  wire  en,
                 input  wire rst );
  reg t;

  assign Q = t;

  initial
  begin
    t = 0;
  end

  always @ ( posedge en )
  begin
    if( rst == 1 )
      t = 0;
    else
      t = D;
  end

endmodule
```

Listing 3.15 An implementation of a D-type flip-flop with reset.

3.4.5 Further Behavioural Statements

3.4.5.1 Conditional Statements

Basic conditional behaviour can be implemented almost identically to in C by using the `if` statement; this takes the form

```
if      ( x )
  statement0;
else if( y )
  statement1;
else
  statement2;
```

To decide which of `statement0`, `statement1` or `statement2` gets executed we first evaluate the expression `x` and, if the value is non-zero, `statement0` is executed. Otherwise, we carry on to the next case, evaluate `y` and test if it is non-zero; if it is, then `statement1` is executed. If neither `x` or `y` evaluates to a non-zero value then `statement2` is executed as the default case. A slight problem is introduced by considering whether the unknown value `1'bX` "means" zero or non-zero. For an `if` statement, this is resolved by defining the unknown value as zero. In this example, if `x` and `y` both evaluate to the unknown value then we end up in the default case where `statement2` is executed. Note that each single statement associated with each test can in fact be a block of statements surrounded with `begin` and `end` keywords.

As an example of the `if` statement in action, consider upgrading our D-type flip-flop design from Listing 3.14 to include a signal which zeros or resets the stored value. This alteration is shown in Listing 3.15 and simply performs a test on the

rst signal each time a positive clock edge triggers the process: if rst is equal to 1, then we set the stored value t to 0, otherwise we set it to the input D.

Long or nested if statements can quickly become hard to maintain as the number of alternatives and complexity of conditions grows. To combat this, Verilog includes a construct called case which is very similar to the switch statement in C. A case statement takes an expression which can evaluate to one of several options; depending on which of these options it does actually evaluate to, one of several statements is executed. It essentially acts like a multi-way or computed branch and takes the form

```
case( x )
  value0  : statement0;
  value1  : statement1;
  value2  : statement2;
  ...
  default : statementN;
endcase
```

The expression x, which can include operators and so on rather than being simply a wire or register, is first evaluated. If the value equals value0, then statement0 is executed; if it equals value1, then statement1 is executed and so on. At most one statement is executed, and there is no need for a break statement as is needed in C. If the value of x does not equal any of the options then the statement corresponding to the default value is executed; if there is no default option then no statements are executed. Note that we can never have more than one default options and that, like an if statement, each single statement associated with a value can be a block of statements surrounded with begin and end keywords.

A similar problem to usage of if statements is presented when considering what happens if x evaluates to the unknown or high impedance value; since the case statement performs a bit-by-bit comparison even a single bit within x which is not 0 or 1 can result in unexpected behaviour. One method to deal with such options is to explicitly list them so that, for example, if we use

```
case( x )
  2'b00   : y = 1'b0;
  2'b01   : y = 1'b1;
  2'b10   : y = 1'b1;
  2'b11   : y = 1'b0;

  2'b0x   : y = 1'bX;
  2'b1x   : y = 1'bX;
  2'bx0   : y = 1'bX;
  2'bx1   : y = 1'bX;

  default : y = 1'b0;
endcase
```

then when x evaluates to the value 2'b0X we execute the fifth statement rather than the default statement as would have happened before. The other way to deal with this problem is to use the casex and casez variations of the basic case construct. A casez statement treats all occurrences of the high impedance value as

don't care; a `casex` treats both the unknown and high impedance values as don't care. This applies to both the case expression (i.e., `x` in the above) or one of the case options (e.g., `2'b0X` in the above). By treating these values as don't care we exclude them from comparison so that, for example, the case value `2'b0X` would match the case option `2'b00` (if not previously matched) when using a `casex` statement instead of a standard `case` statement. Likewise, the case value `2'b00` would match the case option `2'bX0` (if not previously matched) when using `casex` instead of `case`.

As an example of the `case` statement in action, recall the traffic light controller we developed previously in the form of a state machine. The idea was that we stored some state and had a function which moved us to a next state on each positive clock edge; we also had an associated function to set the value of various traffic lights based on the state. Listing 3.16 sketches an implementation of this design, omitting some of the cases for brevity. The module holds the 3-bit current state in the register `S` and initialises this to zero using an `initial` block. On every positive clock edge the first `always` process is triggered; we evaluate `S` and update the value to match the required next state. Also, each time `S` changes we trigger a second `always` block which again evaluates `S` but this time sets the value of output wires corresponding to the traffic lights.

3.4.5.2 Iteration Statements

It is often useful to iterate or loop over a block of statements a given number of times. Although the `if` and `case` statements allow us to control the flow of execution, they are unsuitable for this task. Likewise, an `always` process offers some form of iteration but it is also unsuitable in the sense that it is too coarse grain: we require a set of statements similar to `do`, `while` and `for` loops available in C programs.

The `forever` loop simply performs an infinite loop over the body of statements associated with it. Although in software this is generally considered a bad idea, in hardware it can make perfect sense: for example, a keyboard waits in an infinite loop for keys to be pressed until it is eventually powered off. A `forever` loop takes the form

```
forever
begin
  ...
end
```

where the continuation dots represent the body of code which is looped over. In common with all iteration statements in Verilog, there is no way to exit from the loop prematurely. In a C loop this is performed using the `break` statement; there is no equivalent in Verilog.

The next step up in terms of complexity is a loop which instead of iterating forever, iterates a fixed number of times. The `repeat` statement, not seen in languages

```
module traffic( input  wire clk,

                output reg   Mr,
                output reg   Ma,
                output reg   Mg,
                output reg   Ar,
                output reg   Aa,
                output reg   Ag );

  reg [2:0] S;

  initial
  begin
    S = 0;
  end

  always @ ( posedge clk )
  begin
    case( S )
      0       : S = 1;
      1       : S = 2;
      2       : S = 3;
      3       : S = 4;
      4       : S = 5;
      5       : S = 6;
      default : S = 0;
    endcase
  end

  always @ ( S )
  begin
    case( S )
      0       : begin
                  Mr = 0; Ma = 0; Mg = 1;
                  Ar = 1; Aa = 0; Ag = 0;
                end

      ...

      5       : begin
                  Mr = 0; Ma = 1; Mg = 0;
                  Ar = 1; Aa = 0; Ag = 0;
                end
      default : begin
                  Mr = 0; Ma = 0; Mg = 0;
                  Ar = 0; Aa = 0; Ag = 0;
                end
    endcase
  end

endmodule
```

Listing 3.16 An implementation of the traffic light controller state machine.

like C, offers exactly this behaviour. For example, if we want to execute a body of statements thirty two times we would use

```
repeat( 32 )
begin
  ...
end
```

Note that the value that determines the number of iterations must be a constant rather than an arbitrary expression. Where an arbitrary number of iterations is required, we use a `while` loop such as

```
while( x )
begin
  ...
end
```

In this case the expression `x` is evaluated and tested before each loop iteration; if the resulting value is non-zero, then iteration continues, if it is zero then the loop terminates.

Finally, we have a `for` loop which offers a powerful but simple way to iterate over the loop body. Definition of a `for` loop uses three expressions: an initialiser, a conditional, and an updater. The idea is that the initialiser expression sets loop counters to an initial value. Before each time the loop body is executed the conditional expression is evaluated: if the resulting value is non-zero the loop is executed, otherwise it terminates. After each loop execution the update expression is evaluated to update the loop counters. An an example which executes a body of statements 32 times we could write

```
for( i = 0; i < 32; i = i + 1 )
begin
  ...
end
```

In this case we have `i = 0` as the initialiser which sets the loop counter `i` to zero; `i < 32` is the conditional which allows the loop to continue as long as `i` is less than 32; the update expression `i = i + 1` increments the loop counter each time the loop is executed. Note that unlike the equivalent statement in C, we cannot use the increment or decrement operators: we are forced to use the expression `i = i + 1`. Also note that in some sense a `for` loop is just a shorthand; we could write the above using just a `while` loop as

```
i = 0;
while( i < 32 )
begin
  ...
  i = i + 1;
end
```

As in the case of `always` blocks, there is a problem if the body of statements one iterates over has no delays in it: again we are saying that every iteration happens at the same time.

3.4.6 Tasks and Functions

We have already drawn analogies between modules in Verilog and functions in C, but have tried hard to show that they are different. Even so, it is useful to have some means of defining shared or common blocks of code rather than replicating that code wherever required. Two methods of achieving this can be used in Verilog; they are called **tasks** and **functions** respectively. In a rough sense, functions can only be combinatorial results while tasks can be fairly arbitrary blocks of code. More exactly, there are some constraints on how tasks and functions can be constructed:

- Tasks can have any number of inputs and outputs; functions must have at least one input and no (explicit) outputs.
- Tasks can contain fairly arbitrary statements including timing control; functions cannot contain timing control statements.
- Tasks can call other tasks and other functions; functions can only call other functions.

All tasks and functions are local to the module in which they are defined and can access all wires and registers defined within that module; neither can contain locally defined wires or registers.

Consider the following example function which calculates the parity of an input, i.e., whether there are an odd or even number of 1 bits set in the value:

```
function parity;
  input [7:0] x;
begin
  parity = ^x;
end
endfunction
```

After giving the function a name or identifier, the 8-bit input `x` is defined. The body of the function simply applies the XOR reduction operator to `x` and forms the return value by assigning this to a pseudo-value with the same name of the function. The function can then be called much like one would in C. The following example task demonstrates this by using the `parity` function to perform a parity check of an input `x` against a parity bit `b`:

```
task parity_check
  input  [7:0]   x;
  input        bit;
  output       err;
begin
  if( parity( x ) == bit )
    err = 0;
  else
    err = 1;
end
endtask
```

If the calculated parity of input `x` is equal to the input check bit called `bit`, then we set the output `err` to 0, otherwise we set it to 1 to mark an error has occurred.

We can then use the task within some block of code. As a slightly more contrived example, consider reading an 8-bit value and an associated check bit from some input source; having obtained all the bits we could use the task to check the parity and perform some operation as a result:

```
reg err;

always @ ( posedge clk )
begin
  x[0] = read_bit();
  x[1] = read_bit();
  x[2] = read_bit();
  x[3] = read_bit();
  x[4] = read_bit();
  x[5] = read_bit();
  x[6] = read_bit();
  x[7] = read_bit();

  bit  = read_bit();

  parity_check( x, bit, err );

  if( err )
    ...
  else
    ...
end
```

If the `parity_check` task is used often enough within the encompassing module, then we will have reduced the amount of replicated code (although not necessarily the logic that implements it) and associated risk of introducing errors. Notice here that the inputs and outputs of the task must be valid in the context of their use. For example, the procedural assignment to `err` in the task means the argument passed in must be a register.

3.5 Effective Development

Like most forms of programming, hardware design using a HDL like Verilog is more than simply knowing the language syntax and semantics. Without some experience actually using the language to produce functioning models, many of the pitfalls and intricacies will remain hidden. With this in mind, the following section attempts to convey some good design practises and features in the Verilog language that can improve the quality of code in terms of maintainability and functional correctness.

`$display( <format>, <args> ... )` Print output to the console.
`$monitor( <format>, <args> ... )` Print output whenever values change.
`$monitoron` Enable all monitor output.
`$monitoroff` Disable all monitor output.
`$stop` Halt the simulator.
`$finish` Exit the simulator.
`<value> = $random( <seed> )` Produce a random value.
`<handle> = $fopen( <filename> )` Open a file for input.
`$fclose( <handle> )` Close an open file.
`$fdisplay( <format>, <args> ... )` Direct `$display` output to file.
`$fmonitor( <format>, <args> ... )` Direct `$monitor` output to file.
`$readmemb( <filename>, <memory>, <start>)` Read ASCII binary file into memory.
`$readmemh( <filename>, <memory>, <start>)` Read ASCII hexadecimal file into memory.

Table 3.8 A list of common system tasks.

3.5.1 System Tasks

Since during simulation a Verilog model is, by definition, running on some form of controlled platform, it makes sense to allow it access to some of the resources on that platform. For example, it is useful to allow a model to issue messages to the console or access the file system to load test data or store results. Although such a facility will not be present in real hardware, the use of a **system task** as a means of accessing functionality on the host platform is a massive aid to the development and debugging cycle.

A system task is similar to a normal Verilog task except that is it executed by the simulator rather than the model. As such, they are somewhat dependent on the exact simulator used although standardised Verilog includes a number of system tasks which should always be available. Table 3.8 lists some common system tasks;

perhaps the most useful are the `$display` and `$monitor` tasks which offer a simple means of debugging.

The `$display` task is similar to the `printf` function in C in that one gives it a format string and a list of values to fill into the template. Using this task, the translated string is printed to the console of the simulator so that it can be read by a user. For example, the following block

```
begin
  x = 10;
  y = 20;

  $display( "x is %b in binary, y is %d in decimal", x, y );
end
```

uses the `%b` formatting parameter to format value `x` as a binary number and the `%d` formatting parameter to format `y` as a decimal value. The resulting string, "x is 1010 in binary, y is 20 in decimal" is printed to the console so that one can verify that `x` and `y` have the values expected. However, the use of many `$display` tasks to track the value of a wire of register can be cumbersome; often we want to just print the value whenever it changes. The `$monitor` task offers a simple way to achieve this. The syntax is similar to a `$display` task in that it takes a format string and a list of values but rather than being executed just once, the monitor task is executed whenever one of the listed value changes. Thus, using the block

```
begin
  $monitor( "x is %b in binary, y is %d in decimal", x, y );

  $monitoron;
  x = 10;
  y = 20;
  $monitoroff;
end
```

we would expect two messages to be printed to the console: one after `x` is assigned to and one after `y` is assigned to.

3.5.2 Using the Pre-processor

Constant values without an obvious meaning in a C program or Verilog program are often called "magic": they make things work but it often is not clear why their specific value is chosen. As a solution to this problem, C allows the use of a **pre-processor** with which one can manipulate the source code before it is fed to the compiler. For example, instances of defined macros are replaced with an associated value; the program source code is hence more easily understood.

Although less sophisticated than the C pre-processor, Verilog allows similar functionality through directives to the tool-chain. Perhaps the most simple example is the `` `include `` directive which allows one to include the contents of a Verilog source file in another. This practise should be used with care so that cycles of inclusion are

```
1  `define WIDTH 4
2
3  module add_nbit( output wire              co,
4                   output wire [`WIDTH-1:0]  s,
5                   input  wire               ci,
6                   input  wire [`WIDTH-1:0]  x,
7                   input  wire [`WIDTH-1:0]  y );
8
9    assign { co, s } = x + y + ci;
10
11 endmodule
```

Listing 3.17 Using pre-processor directives to replace global constants.

```
1  `define GATES
2
3  module add_1bit( output wire co,
4                   output wire  s,
5                   input  wire ci,
6                   input  wire  x,
7                   input  wire  y );
8
9  `ifdef GATES
10   wire w0, w1, w2;
11
12   xor t0( w0, x,  y  );
13   and t1( w1, x,  y  );
14
15   xor t2(  s, w0, ci );
16   and t3( w2, w0, ci );
17
18   or t4( co, w1, w2 );
19 `else
20   assign { co, s } = x + y + ci;
21 `endif
22
23 endmodule
```

Listing 3.18 Using pre-processor directives for conditional source code inclusion.

avoided and is best used, for example, to include a single file of macro definitions that are common to the entire model.

Continuing the similarity with the C pre-processor, the `` `define `` directive allows the definition of a macro which is replaced the associated value when used subsequently. Listing 3.17 shows the use of `` `define `` in action by implementing an adder whose input and output ports are wider than 1-bit. By altering the definition of the macro `WIDTH`, we can change how wide these ports are without fiddling with the main module definition. Note that when using a macro in C, one would only need to use the macro identifier `WIDTH` while in Verilog we are required to prefix the identifier with a quote to get `` `WIDTH ``.

Conditional inclusion or exclusion of Verilog code can be achieved using the `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `elsif `` and `` `endif `` directives. This sort of construct can alter the source code fed to the Verilog tool-chain by simply changing some defined constants rather than major alterations to the source code itself. List-

ing 3.18 demonstrates an example; we have two different implementations of the adder module and select which one is used in the design by simply defining or not defining the `GATES` macro. Code for whichever branch is taken is fed to the tool-chain, the branch which is not taken is omitted as if it were never there or simply commented. Since macros can be defined by the tool-chain as well as in the source code, conditional inclusion and exclusion of code can offer a neat way to configure a design for different purposes and situations.

The `‘timescale` directive is a means of changing the unit of time during simulation. This alters the units used to define timing delays without the need to scale them all by hand, a daunting task in a large design. Use of `‘timescale` follows the general format:

```
‘timescale <reference> / <precision>;
```

where the `<reference>` field determines the unit of measurement while the `<precision>` field specifies the precision to which delays are rounded. There are some constraints as to what values can be used in both fields but these are largely dependent on the simulator.

3.5.3 Parameters

Although using pre-processor directives to ease maintainability is a useful idea, often they are not powerful enough to achieve what we want. For example, consider the definition of an adder module with variable sized input and output ports as described in Listing 3.17. By using the macro `WIDTH` we can easily change the size of the ports but this is a global change: *all* `add_nbit` modules will have the *same* sized ports, what if we want one with 4-bit ports and another with 2-bit ports ?

Module **parametrisation** solves this problem by allowing the declaration of parameters for each module which can be overridden by each instance. The `parameter` keyword is used to declare a parameter with a default value. Essentially similar to a macro definition, the parameter symbol can then be used freely within the module body, although it cannot be assigned to since it is not a variable. The major difference between using a parameter is that each instance can set it to a different value. This is shown in Listing 3.19 where we parametrise the adder module and then form two instances. The instance named `t0` has the `WIDTH` parameter set to four so that the ports are four bits wide; The instance named `t1` has the `WIDTH` parameter set to two so that the ports are two bits wide. This is performed using the `defparam` keyword and utilises the Verilog hierarchical naming scheme to specify exactly which instance and parameter are being referred to.

Note that we mentioned in Section 3.2 that the form of module definition where the ports are typed inside the module body is sometimes useful: parameters are one reason why. Since the parameter is defined inside the body, one cannot use the symbol before it is defined and hence the parameter symbol cannot exist in the module header.

```
1  module add_nbit( co, s, ci, x, y );
2
3    parameter WIDTH = 0;
4
5    output wire             co;
6    output wire [WIDTH-1:0] s;
7    input  wire             ci;
8    input  wire [WIDTH-1:0] x;
9    input  wire [WIDTH-1:0] y;
10
11   assign { co, s } = x + y + ci;
12
13 endmodule
14
15 module top();
16
17   wire       t0_co, t0_ci;
18   wire       t1_co, t1_ci;
19
20   wire [3:0] t0_s, t0_x, t0_y;
21   wire [1:0] t1_s, t1_x, t1_y;
22
23   add_nbit t0( t0_co, t0_s, t0_ci, t0_x, t0_y );
24   add_nbit t1( t1_co, t1_s, t1_ci, t1_x, t1_y );
25
26   defparam t0.WIDTH=4;
27   defparam t1.WIDTH=2;
28
29 endmodule
```

Listing 3.19 A full-adder implementation whose port width is parametrised.

```
1  add_1bit t0(     w0 ,    w1 ,     w2 ,    w3 ,    w4  );
2  add_1bit t1( .co(w0), .s(w1), .ci(w2), .x(w3), .y(w4) );
3  add_1bit t2( .co(w0), .s(w1), .ci(w2), .y(w4), .x(w3) );
4  add_1bit t3(          .s(w1),          .x(w3), .y(w4) );
```

Listing 3.20 Using named port lists to instantiate a full-adder.

3.5.4 Named Port Lists

As modules get more complex, the number of input and output ports typically grows quite large. This presents a number of problems. Firstly, it becomes difficult to remember the order the ports are declared in and one can easily wrongly connect two modules as a result. Secondly, it is quite common to want to omit specific connections to complex modules because one does not use the functionality they provide. For example, we might ignore the write port of a memory module if the memory is used in a read-only mode. In this case, we would like to connect the port to null somehow.

Named port lists solve both of these problems simultaneously. Consider having to instantiate the full-adder module described in Section 3.2 and Section 3.3. Listing 3.20 shows four different ways to do this starting with the conventional method in Line 1. In fact, the first three lines are equivalent; even though the order or some

```
1  module mux2_nbit( r, i0, i1, s0 );
2
3    parameter WIDTH = 0;
4
5    output wire [WIDTH-1:0] r;
6    input  wire [WIDTH-1:0] i0;
7    input  wire [WIDTH-1:0] i1;
8    input  wire             s0;
9
10   assign r = ( s0 ? i1 : i0 );
11
12 endmodule
13
14 module top();
15
16   wire [31:0]  r;
17   wire [31:0] i0;
18   wire [31:0] i1;
19   wire        s0;
20
21   mux2_1bit t0( .r(r), .i0(i0), .i1(i1), .s0(s0) );
22
23   defparam t0.WIDTH = 32;
24
25 endmodule
```

Listing 3.21 A parametrised implementation of a 1-bit, 2-way multiplexer with 32-bit ports.

wires in Line 2 and Line 3 are different, the fact we name the ports we want to connect to ensures the instantiations are correct. The final instantiation in Line 4 omits the carry-in and carry-out ports. The carry-in input is therefore wired to zero for us by the Verilog tool-chain, while the carry-out output is simply ignored or left dangling.

3.5.5 Generate Statements

Consider the task of creating a 2-way multiplexer where the inputs and outputs are no longer 1-bit values as previous discussed, but 32-bit values. One way to go about this is clearly to simply utilise wire vectors and increase the width of the ports. We might even use parameters as shown in Listing 3.21.

Another way to do the same thing is to use a Verilog **generate statement**. A generate statement is designed to programatically and automatically generate hardware instances; the statement is a means of manipulating the design rather than something which is executed at run-time. As such, one can think of it as another form of directive. The idea is that instead of typing out thirty two instantiations of the 2-way multiplexer with 1-bit inputs, we let the Verilog tool-chain do the work for us. Listing 3.22 demonstrates the generate statement in action; the generate block encloses a `for` loop which the tool-chain executes to instantiate thirty two multiplexers, each one taking the *i*-th bit of the two inputs and producing the *i*-th bit of the result. Note that in this case, Verilog demands our loop counter is declared using the `genvar`

```
1  module mux2_1bit( output wire  r,
2                    input  wire i0,
3                    input  wire i1,
4                    input  wire s0 );
5
6    assign r = ( s0 ? i1 : i0 );
7
8  endmodule
9
10 module top();
11
12   wire [31:0]  r;
13   wire [31:0] i0;
14   wire [31:0] i1;
15   wire        s0;
16
17   genvar i;
18   generate
19     for( i = 0; i < 32; i = i + 1 )
20     begin:gen_mux
21       mux2_1bit t( .r(r[i]), .i0(i0[i]), .i1(i1[i]), .s0(s0) );
22     end
23   endgenerate
24
25 endmodule
```

Listing 3.22 A generate statement building a 1-bit, 2-way multiplexer with 32-bit ports.

type to distinguish it from types used for statements which need to be executed at run-time.

Looking at the two solutions, the parametrised version seems more sensible; certainly it does the right job and also promotes reuse of the module. However, there is a subtle design principle underlying the use of generate which goes beyond simply reducing work. The Verilog tool-chain is good at taking models and running simulation and synthesis processes on them. But like any software, it is finite in computational power and generally fallible in the face of complex inputs. In fact, a Verilog tool-chain will typically do a much better job of any optimisation when faced with simple, small constructs rather than large ones. With this in mind, creating models from lots of modules that operate on small inputs is often more desirable than using one module that operates on a single large input.

3.5.6 Simulation and Stimuli

One of the major advantages of using Verilog over raw hardware is that one can simulate and debug a circuit before it is sent through the expensive process of manufacturing. Roughly speaking, there are two types of Verilog simulator:

Cycle-based Simulator The most simple form of simulator assumes that the entire model is clocked and that change occurs synchronously at clock edges. The

simulator runs a clock and on each transition calculates the new state of the model; any changes in between the clock transitions are lost.

Event-based Simulator Roughly speaking an event-based simulator waits, watching for changes in any values within the model and then propagates those changes to elements they effect. Events occur relative to an internal clock which tracks the progress of simulation time; the current time is typically stored in a variable named `$time` which is accessible for debugging purposes.

For example, the simulator might watch the inputs to a module and, when they change, evaluate expressions to compute the output or trigger execution of a process. Complex scheduling methods determine the order such events are processed in so that simulation remains valid and to reduce the amount of computation required.

In some sense, the difference in approach represents a trade-off between simulation speed and accuracy although modern simulators of either type are effective. They can be further sub-divided into interpreted simulators versus those which perform some form of compilation on a model before simulation. Purely interpreter-based systems process the model line-by-line and as a result are slower. In contrast, compilation-based simulation translates the model into an internal, easier to deal with format before executing it. This has the advantage that the simulation can be much faster; optimisation of the internal format and a global view of the model can give significant performance benefits. An extreme example of a compiler-based simulator might translate the model into native executable code for the host machine. Obviously the drawback is that every time the model changes, it must be recompiled before simulation can restart so the internal representation matches the design.

None of the modules we have looked at thus far have been self-contained: they include input and output ports to external systems. In part it is obvious that any useful module will look like this since if it takes no input and produces no output as a side effect, we can optimise the design by replacing it with the empty module ! This sort of module with no inputs and no outputs is usually called a top-level module since it is self-contained, at the top of the hierarchy. To follow convention, we usually call the top-level module `top`; clearly it can be called anything.

No matter what simulation system is used, to simulate a model we typically need to feed it test signals and collect the outputs. This is the job of a top-level stimulus module. Such a module encompasses the design, instantiating an instance of it and stimulating the inputs in order to provoke execution. For example the stimulus might toggle the reset signal to jump-start the design, emulating the behaviour of the physical switch one would wire onto the design when implemented in real hardware.

Figure 3.3 describes diagrammatically how a stimulus for our full-adder interfaces with the module. The stimulus module instantiates the full-adder and connects the ports to nets internal to it. These nets can then be set to values in order to provoke the full-adder to produce output which can be inspected in the simulator. Listing 3.23 demonstrates an implementation of this: we instantiate the adder and then use a single process to toggle `ci`, `x` and `y` through all possible values to test the module. To verify that we are getting the right results on `co` and `s`, we use the

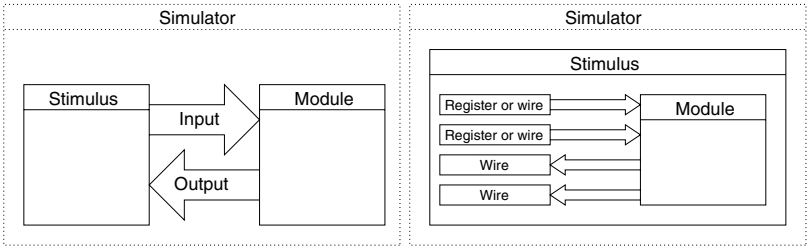


Figure 3.3 A stimulus module encompasses the design and feeds it test values.

```
1  module top();
2
3    wire co;
4    wire  s;
5    reg  ci;
6    reg   x;
7    reg   y;
8
9    add_1bit t0( .co(co), .s(s), .ci(ci), .x(x), .y(y) );
10
11   initial
12   begin
13     $monitor( "%b %b %b %b %b", co, s, ci, x, y );
14
15     #10 ci = 0; x = 0; y = 0;
16     #10 ci = 0; x = 0; y = 1;
17     #10 ci = 0; x = 1; y = 0;
18     #10 ci = 0; x = 1; y = 1;
19     #10 ci = 1; x = 0; y = 0;
20     #10 ci = 1; x = 0; y = 1;
21     #10 ci = 1; x = 1; y = 0;
22     #10 ci = 1; x = 1; y = 1;
23   end
24
25 endmodule
```

Listing 3.23 Implementation of a stimulus module for a full-adder.

`$monitor` system task to track their values as they change. The use of `$monitor` reduces the amount of typing we do but increases the verbosity of the output; we might also have used `$display` to inspect the values at specific points in execution.

3.6 Further Reading

- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.

- S. Palnitkar.
  Verilog HDL: A Guide in Digital Design and Synthesis.
  Prentice-Hall, 2003. ISBN: 0-130-44911-3.
- D.E. Thomas and P.R. Moorby.
  The Verilog Hardware Description Language.
  Springer-Verlag, 2002. ISBN: 1-402-07089-6.

3.7 Example Questions

12. Write the following as Verilog declarations:

a. An 8-bit little-endian wire vector called `a`.
b. A 5-bit big-endian wire vector called `b`.
c. A 32-bit register called `c`.
d. A signed 16-bit register called `d`.
e. A memory of 1024 elements, each 8 bits in size, called `e`.
f. A generate variable called `f`.

13. Given the declarations:

```
wire [3:0] a;
wire [3:0] b;
wire [1:0] c;
wire [3:0] d;
wire       e;

assign a = 4'b1101;
assign b = 4'b01XX;
```

what are the values resulting from the following assignments:

a. `assign c = a[1:0];`
b. `assign c = a[3:2];`
c. `assign d = a & b;`
d. `assign d = a ^ b;`
e. `assign d = { a[3:2], a[1:0] };`
f. `assign d = { a[1:0], a[3:2] };`
g. `assign c = {2{b[1]}};`
h. `assign c = {2{b[2]}};`
i. `assign e = &a;`
j. `assign e = ^a;`

14. a. Consider the following Verilog processes for appropriately defined 1-bit wires `a`, `b`, `x`, `y`, `p` and `q`:

```
always @ ( posedge p )
begin
  x <= a;
  y <= b;
end

always @ ( posedge q )
begin
  x <= b;
  y <= a;
end
```

Given that `p` and `q` are independent and may change at any time, write down **one** potential problem with this design and outline **one** potential solution.

b. Consider the following Verilog process, for appropriately defined 1-bit wire `clk` and 2-bit wire vector `state`, which implements a state machine with three states:

```
always @ ( posedge clk )
begin
  case( state )
    0 : begin
          do_0;
          state = 1;
        end
    1 : begin
          do_1;
          state = 2;
        end
    2 : begin
          do_2;
          state = 0;
        end
end
```

If this process constitutes the entirety of the design, write down **one** potential problem with it and outline **one** potential solution.

15. A Decimal Digit Detector (DDD) is a device that accepts a 1-bit input at each positive clock edge, and waits until four such bits have been received. At this point, it sets a 1-bit output signal to true if the 4-bit input (interpreted in little-endian form) is a valid Binary Coded Decimal (BCD) digit and false if it is not; it then repeats the process for the next set of four input bits.

Design a Verilog module to model the DDD device. Your design should incorporate a reset signal that is able to initialise the DDD.

16. A comparator C is a function which takes two unsigned n-bit integers x and y as input and produces $\max(x,y)$ and $\min(x,y)$ as outputs. One can think of C as sorting the 2-element sequence (x,y) into the resulting sequence $(\min(x,y),\max(x,y))$. Design a Verilog module to model a single comparator for n-bit numbers.

17. An n-bit Linear Feedback Shift Register (LFSR) called R is a register whose n-bit content is shifted right on each positive clock edge; if we use R_i to refer to the i-th bit of R, this means that R_0 is shifted out of the register then provided as output from the LFSR, and we need to create a new bit to replace R_{n-1}. The new bit is created by XORing together bits from the register according to a tap sequence; if the tap sequence is $0,2,3,5$ then the new value for R_{n-1} shifted in is $R_0 \oplus R_2 \oplus R_3 \oplus R_5$. Design a Verilog module to model an 8-bit LFSR with the tap sequence $3,5,7$. Your design should incorporate a reset signal that is able to initialise the LFSR with a seed value given as input.

Part II
Processor Design

Chapter 4
A Historical and Functional Perspective

Computers are good at following instructions, but not at reading your mind.

– D. Knuth

Abstract If one looks back through history, the terms computer processor and computer are somewhat synonymous; they are both used refer to devices that perform some computation. Today we typically separate the two concepts and use computer to refer to the entire system, including memory, input/output peripherals and so on. The term processor is roughly used to refer to the **Central Processing Unit (CPU)** which is normally realised on a single microchip. The CPU is at the heart of the computer, the collection of logic it houses is tasked with executing programs that generate results for the user. Our aim in this chapter is to survey the development and introduce the function of a basic CPU. That is, the aims of this chapter do not relate to the implementation of the CPU in digital logic: we focus almost entirely on the concepts and mechanisms that dictate how such a CPU will execute programs.

4.1 Introduction

This general subject area has a rich and interesting history recorded in detail by a large number of dedicated, knowledgeable authors. Throughout the chapter we make citations to specific works of interest; an interested reader can find a more thorough treatment in the books of Brian Randell [56] and Paul Ceruzzi [12], and in the extensive IEEE Annals of the History of Computing journal or ACM History of Programming Languages (HOPL) conference.

It might seem a simplistic comparison, but the design and use of a computer relates well to how one might design and use a pocket calculator which, at face value, seems far more primitive:

A pocket calculator:

- has an **accumulator** (i.e., current value) and some **memory** (accessed via the $M+$ and MR buttons) that can store values.
- has some component that performs arithmetic (e.g., add together numbers).
- has input and output **peripherals** (e.g., keypad, LCD screen).
- responds to simple commands or **instructions** from the user; for example, the user can supply things to perform arithmetic on (i.e., numbers) and commands to perform arithmetic (e.g., do an addition).

A computer processor:

- has many **accumulators** (or **registers**), and potentially many levels and large amounts of **memory**.
- has an **Arithmetic and Logic Unit (ALU)** to perform arithmetic.
- has input and output **peripherals** (e.g., hard disk, monitor).
- executes sequences of simple instructions called programs; instructions are made up from **operands** that define values to operate on, and **opcodes** which define the operation to perform.

In reality, a pocket calculator is a type of special-purpose computer so the similarity is probably not surprising; unlike what we may traditionally view as a computer, the calculator exists to perform the single task of arithmetic on behalf of the user. From the perspective of someone using a computer however, the analogy is not always as obvious. In particular, this is because features such as the user interface, operating system and programming languages mask the simplicity of low-level behaviour.

In terms of teaching and learning about computer science, this presents a major problem: on one hand languages such as Java offer a powerful means to construct software, but on the other hand programmers using Java are many levels away from the raw hardware that executes the end result. This chapter represents a starting point toward explaining how a computer *actually* executes programs and thereby allows it to be generally useful rather than fixed to some special task such as arithmetic. The aim is to overview evolutionary steps using examples from history to highlight key advances, and to demonstrate at a functional level what typical low-level behaviour is.

4.2 Special-Purpose Computers

Ignoring the era of mechanical computers such as the famous Differential and Analytical Engine designs of Babbage, there are a number of milestones that demand a brief overview. One of the first machines we would recognise as a computer today was the **Colossus** [14, 18, 23], often cited as the first electronic computer. Although described as a programmable computer, Colossus was built by British code-breakers for a single purpose: to decipher messages encrypted by the German-designed **Lorenz machine** during the Second World War. The first, prototype Colossus contained 1500 electronic valves (essentially vacuum tubes) and was brought

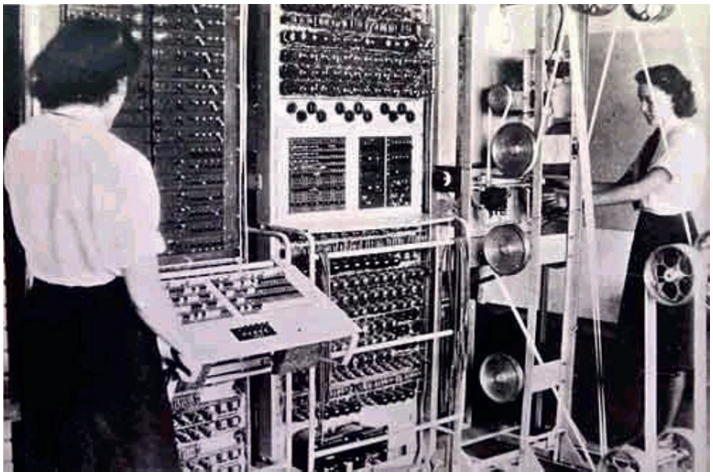


Figure 4.1 Operators using the Colossus computer and paper tape reader (Public domain image, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:Colossus.jpg`).

on-line at Bletchley Park in early 1944; ten such machines were built and used by the end of the war. Colossus was designed and built by a team led by engineer Tommy Flowers, initially at the Post Office Research Station, after weaknesses in the Lorenz cipher were found by Bill Tutte, a cryptanalyst at Bletchley Park. In a technical sense, the computer was designed to apply a programmable Boolean function to inputs read from a paper tape at high speed. By counting how often the Boolean function was evaluated as true, Colossus mechanically unpicked statistical features in message tapes that enabled their eventual decryption.

Lorenz was not the only cipher used by the Germans; the rotor-based **Enigma machine** is perhaps more famous and offered at least an equal challenge to Allied forces. Using Polish cryptanalytic techniques, Bletchley Park developed an automated way to decipher messages secured by Enigma. Put very simply, the purpose of the so-called **Bombe** computer was to emulate the rotor structure of Enigma and search different configurations until a statistical test was passed; this was done at very high speed. The correct configuration was deemed found when an output that statistically represented natural language was detected. The first devices, built under the direction of Harold Keen following initial designs by Alan Turing, were installed in 1940. By 1945 roughly two hundred machines were being operated by staff in order to decipher German communication.

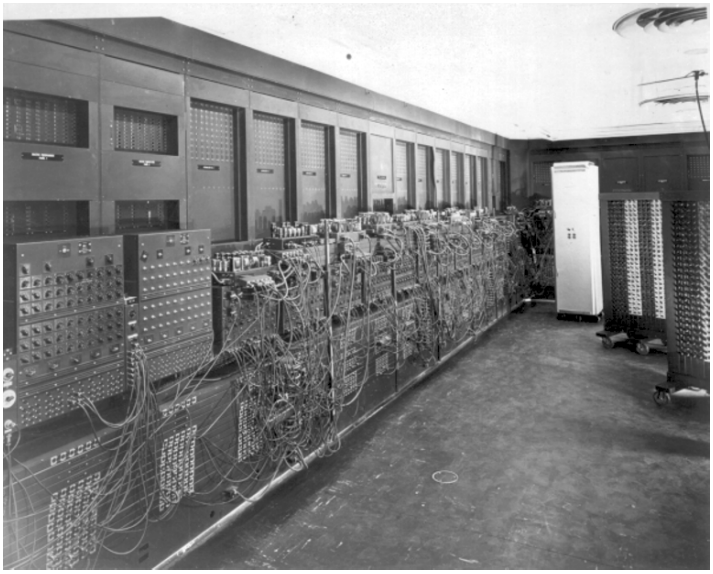


Figure 4.2 The ENIAC installation at the U.S. Army Ballistics Research Laboratory. (A U.S. Army photograph now in public domain, photographer: Unknown, source: `http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac5.gif`)

4.3 General-Purpose Computers

Although the Colossus and Bombe were partly programmable in the sense that their behaviour could be altered by reconfiguring the physical machine using a plugboard, they were not general-purpose in the way we expect today. For a computer to be general-purpose we now say it must be **Turing-complete**, a concept introduced by Alan Turing to capture the notion that a truly general purpose computer be able to compute anything actually computable; it should have a computational power equivalent to a universal Turing machine.

A step toward this goal was the **Electronic Numerical Integrator And Computer (ENIAC)** [9], a computer designed by John Presper Eckert and John Mauchly while working as researchers at the University of Pennsylvania. Although it was designed as a general-purpose computer, the ENIAC was commissioned by the U.S. Army for their Ballistics Research Laboratory; the chief aim was to accurately compute numerical tables used to guide firing of munitions. Until then, the tables were calculated by hand and were difficult to produce within an acceptable accuracy; without said accuracy the risk of friendly fire incidents was presumably significant! The ENIAC was brought on-line in 1946 having cost around $\$500,000$, occupying an entire room and weighing around three tonnes; it used a huge number of electrical components including almost $19,000$ electronic valves, more than ten times as

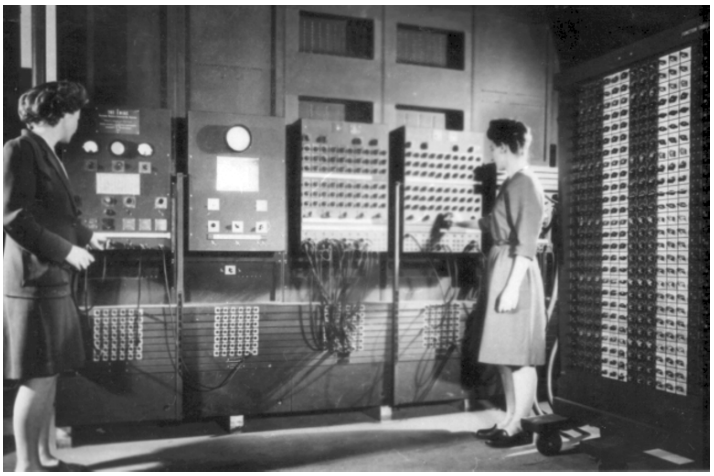


Figure 4.3 Two operators at the ENIAC control panel. (A U.S. Army photograph now in public domain, photographer: Unknown, source: `http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac7.gif`)

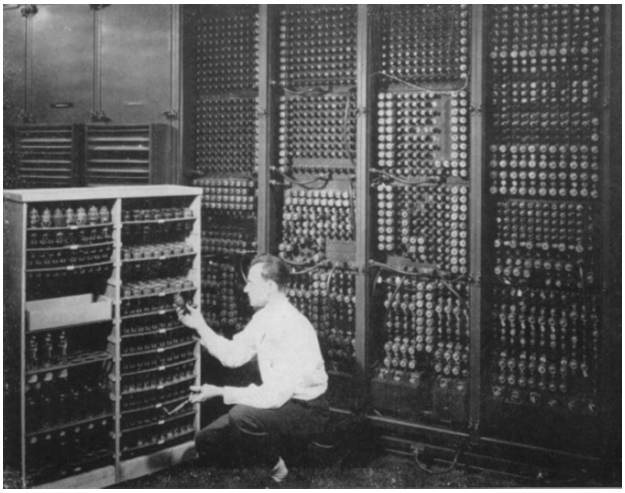


Figure 4.4 Original caption notes that "replacing a bad [vacuum] tube meant checking among ENIACs 19,000 possibilities"; a daunting task ! (A U.S. Army photograph now in public domain, photographer: Unknown, source: `http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac3.gif`)

many as the Colossus. The ENIAC was programmed by configuring the many physical switches and cables that controlled behaviour. In this sense, the ENIAC was programmable and could perform a range of arithmetic operations including addition, subtraction, multiplication and even square root using decimal values from ten-digit accumulators.

Essentially the ENIAC was an early pocket calculator, although one would need large pockets to carry it around ! Early computers of this type viewed instructions (the commands to the computer) and data (the values the commands operated on) as different things; this approach is now termed a **Harvard architecture** after the Harvard Mark 1 computer, a precursor to the ENIAC designed by Howard Aiken circa 1944. Consider an imaginary computer of this era whose programs were fed to it on a paper tape. Like the pocket calculator, we can identify and start to explain the key components in some detail:

Accumulators When using a calculator and entering a value, say 10 for example, with which to perform some computation, it needs to be stored somewhere. One can think of the accumulator as a small area of short-term storage for values while we are working with them. In contrast to a memory which stores many values, the accumulator typically stores just a single value; whereas the values in the memory are potentially long lived, the value in the accumulator is transient and only useful in the short term.

Memory A memory is simply a storage device in which one can store values; one can think of it as an infinite array MEM which can store n-bit binary vectors. As such, one can think of the memory abstractly as two functions

$$\begin{array}{ll} \textsc{Load}(x) & : \{0,1\}^* \to \{0,1\}^n, \qquad \textsc{Load}(x) = MEM[x] \\ \textsc{Store}(x,y) & : \{0,1\}^* \times \{0,1\}^n \to \perp, \quad \textsc{Store}(x,y) = MEM[x] \leftarrow y. \end{array}$$

We call x the **address** or **index**; it is used to select the element in MEM whose value we wish to either retrieve using the LOAD function, or update with y using the STORE function. Of course, in reality we cannot hope to build an infinitely large memory. We bound the size of MEM to say m locations; addresses used to access it are then bit-vectors of length $\lceil \log_2(m) \rceil$.

In terms of our pocket calculator, it is common to find a single location memory that is used to assist calculation. For example, to compute the result of $\frac{10-20}{7+3}$ one might first compute $7+3$ and store this in the memory. Then we can compute $10-20$ and divide the result by a value loaded from the memory. Of course, the analogy is not perfect here since with a single location memory we do not really need an address; the location we are referring to is implicit. The LOAD and STORE functions for the calculator are commonly denoted MR and $M+$ for "memory recall" and "memory add".

Arithmetic and Logic Unit (ALU) Once we can store values in the long and short term, we need a device to perform computation on them. The ALU represents such a device and lies at the heart of any computer processor design, including our example. When the user provokes the addition of two values using

the + and = keys on the calculator, it is the ALU which takes them and calculates the answer.
Typically the ALU is tasked with performing arithmetic on values and comparisons between them. Abstractly, an ALU can be thought of as a collection of functions which take values, normally termed **operands**, and produce results. The function applied to the operands is determined by an **opcode**. For example, imagine we have a family of four functions that operate on n-bit operands and produce n-bit results:

$$\begin{aligned} f_+(x,y) &: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n, \quad f_+(x,y) = x+y \\ f_-(x,y) &: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n, \quad f_-(x,y) = x-y \\ f_\cdot(x,y) &: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n, \quad f_\cdot(x,y) = x \cdot y \\ f_/(x,y) &: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n, \quad f_/(x,y) = x/y \end{aligned}$$

then we can define a simple function that describes a limited ALU as

$$\mathrm{ALU}(x,y,z) : \{0,1\}^n \times \{0,1\}^n \times \{+,-,\cdot,/\} \to \{0,1\}^n, \;\; \mathrm{ALU}(x,y,z) = f_z(x,y)$$

That is, we take the operands x and y and use the opcode z to select which function to apply to them in order to produce the result. So if we want the result of $10-20$ then we invoke the ALU using $\mathrm{ALU}(10,20,-)$ in order to calculate it for us via a call to the function $f_-(10,20) = 10-20 = -10$.

For our imaginary computer, we define one accumulator called A and a memory called MEM each of which can store values represented as six-digit *decimal* numbers. Obviously we have already seen that binary offers a better way to connect the concepts of numbers and circuits for their manipulation; we will use decimal here firstly for clarity, and secondly because computers of this era (including ENIAC for example) did the same. Such a computer would execute instructions in a program using a repeated series of steps which one can view as a simple state machine:

1. Encode a program onto punched tape; load this tape into the tape reader and start the computer.
2. Using the tape reader, fetch the next instruction and call it IR.
3. If the end of tape marker is encountered or IR is the $HALT$ instruction then halt the computer, otherwise execute the instruction IR (i.e., perform the operation it specifies).
4. Repeat from step 2.

Note that the roles of data and instructions are clearly separated; programs were written onto the tape at compile-time but fixed at run-time, one cannot change the tape content once execution is started. To actually write such programs in the first place, we need to define an **instruction set** which details the instructions the computer can understand and execute. A very limited instruction set might include the following:

- $A \leftarrow n$, i.e., load the value n into the accumulator A.

- $MEM[n] \leftarrow A$, i.e., store the value in the accumulator A into address n of the memory.
- $A \leftarrow MEM[n]$, i.e., load the value from address n in memory into the accumulator A.
- $A \leftarrow A + MEM[n]$, i.e., add the value from address n of the memory to the accumulator A and store the result back in the accumulator.
- $A \leftarrow A - MEM[n]$, i.e., subtract the value from address n of the memory to the accumulator A and store the result back in the accumulator..
- $HALT$, i.e., halt or stop execution.

Clearly the instruction set limits the sorts of program we can write; for example, like the Harvard Mark 1 there are no branch instructions so we can only execute instructions on the tape in same sequence they were written. More simply, if we want a "loop" in the program, we need a physical "loop" of tape ! Consider a simple example program which adds together two numbers held in $MEM[4]$ and $MEM[5]$ and then stores the result back into $MEM[6]$; the program consists of the four instructions

$$A \leftarrow MEM[4]$$
$$A \leftarrow A + MEM[5]$$
$$MEM[6] \leftarrow A$$
$$HALT$$

The steps involved in executing the program (and the tape and memory content at each step) are detailed in Figure 4.5 and Figure 4.6. After eight steps, we end up with the result of the addition stored in $MEM[6]$: not exactly earth-shattering news by modern standards, but probably quite an achievement in the 1940s !

4.4 Stored Program Computers

Although it was operational until 1955 the ENIAC was upgraded at various points to improve the memory size and arithmetic performance. The most notable improvement, however, came in 1948 when it was improved to incorporate ideas of John von Neumann who, along with Eckert and Mauchly, had designed a successor to ENIAC called the **Electronic Discrete Variable Automatic Computer (EDVAC)**. von Neumann was a Hungarian mathematician responsible for a huge range of results in computer science, physics, mathematics and economics. Many aspects of the EDVAC were based on a 1945 report by von Neumann [48]. The machine was again commissioned by the U.S. Army Ballistics Research Laboratory and was brought on-line in 1949. The success of the machine, and the general architectural design that it was based on, resulted in Eckert and Mauchly leaving academia to found the Eckert-Mauchly Computer Corporation.

The key advance was the realisation that instructions that control the computer and the data they operate on are both essentially numbers. This is easy to see now: we could for example associate the number 1 with the addition operation and easily

CPU	
state	start
IR	
A	000000

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000000
7	000000

Step t_0 starts by initialising the processor and loading the program onto the tape.

CPU	
state	fetch
IR	$A \leftarrow MEM[4]$
A	000000

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000000
7	000000

Step t_1 fetches instruction #0 from the program tape.

CPU	
state	execute
IR	$A \leftarrow MEM[4]$
A	000010

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000000
7	000000

Step t_2 executes instruction #0, loading the value 10 from memory address 4 into A.

CPU	
state	fetch
IR	$A \leftarrow A + MEM[5]$
A	000010

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000000
7	000000

Step t_3 fetches instruction #1 from the program tape.

CPU	
state	execute
IR	$A \leftarrow A + MEM[5]$
A	000030

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000000
7	000000

Step t_4 executes instruction #1, adding the value 20 from memory address 5 to A.

Figure 4.5 Execution of a simple program on a Harvard-style computer.

CPU	
state	= fetch
IR	= $MEM[6] \leftarrow A$
A	= 000030

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000000
7	000000

Step t_5 fetches instruction #2 from the program tape.

CPU	
state	= execute
IR	= $MEM[6] \leftarrow A$
A	= 000030

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000030
7	000000

Step t_6 executes instruction #2, storing the value 30 in memory address 6.

CPU	
state	= fetch
IR	= *HALT*
A	= 000030

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000030
7	000000

Step t_7 fetches instruction #3 from the program tape.

CPU	
state	= execute
IR	= *HALT*
A	= 000030

Tape	
Address	*Instruction*
0	$A \leftarrow MEM[4]$
1	$A \leftarrow A + MEM[5]$
2	$MEM[6] \leftarrow A$
3	*HALT*
4	
5	
6	
7	

MEM	
Address	*Value*
0	000000
1	000000
2	000000
3	000000
4	000010
5	000020
6	000030
7	000000

Step t_8 executes instruction #3, halting the computer.

Figure 4.6 Execution of a simple program on a Harvard-style computer.

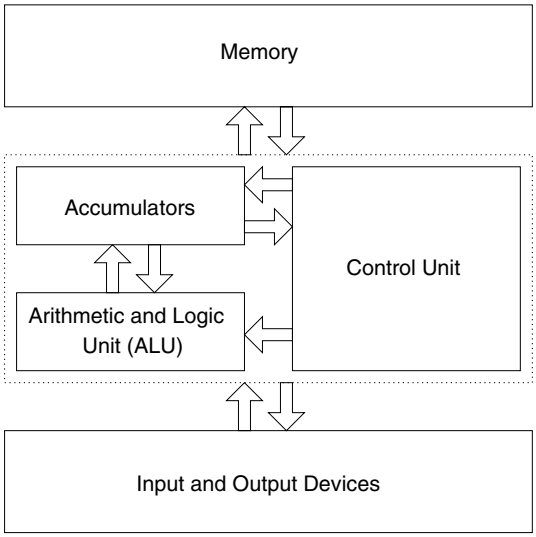


Figure 4.7 The canonical stored program architecture.

store this in memory alongside the data values, the meaning of 1 becoming a simple matter of interpretation. There is certainly some prior art to the ideas of von Neumann, for example Konrad Zuse described the concept circa 1936 in a patent application for his range of early computers and Alan Turing described a complete stored program computer, the Pilot ACE, circa 1946. However, the successful implementation of the so-called **von Neumann architecture** or **stored program architecture** in the EDVAC has become a milestone innovation.

A canonical stored program architecture is comprised of a number of components as shown in Figure 4.7. This diagram makes explicit a number of additional components which were implicit in our original imaginary computer. In particular, we see that the architecture includes:

Control Unit In order to synchronise and control all other aspects of the system, the control unit typically acts as a state machine which steps through a series of operations to perform computation; as such it has implicit hooks into all other devices which it uses to control their behaviour.
Inside a calculator for example, the control unit has a number of tasks: accept input from the user, invoke the ALU where required, display output to the user. Although the user cannot see this happening, the control unit imposes an order on how the calculator can be used and how the devices inside it are used.

Buses The lines that connect components in Figure 4.7 depict channels of communication, for example the ALU communicates information to and from the accumulator. Naively, we can think of these connections as wires that link the

components. However, several factors mean we also need a protocol or policy to manage how the wires are used. For example, both ends of the communication link sending a signal at the same time along the same wire will cause problems that need to be managed.

The general term for such connections is a **bus**; this differs from a point-to-point link in that the communication medium can be shared between several devices if correctly managed by a **bus protocol**. Again, there is no obvious analogy in the context of a calculator. Hopefully it is clear that, inside the device at least, there needs for be communication between the ALU doing computation and the LCD screen output device for example.

Input and Output Devices Input and output devices are the easiest to explain by analogy since they are usually the most obvious in a physical sense. With a calculator, the keypad used to enter values and the LCD screen used to view those values are examples of such devices. Input devices are used by the user to communicate information into the processor, output devices are used by the processor to communicate information to the user.

Considering the overall architecture, we have effectively identified two main subsystems: the **data-path** which is responsible for doing computation and the **control-path** which is responsible for telling the data-path what to do and when to do it. From a programmer's point of view, the capabilities of a processor are more abstractly described by an **Instruction Set Architecture (ISA)**. The ISA specifies how instructions should be formatted so that the control unit can correctly interpret them, and the meaning of those instructions in terms of their **semantics**. As such, an ISA represents an interface between programmer and an abstract processor rather than an exact specification of how the processor should be implemented. For example, the MIPS32 ISA specifies a fairly standard instruction set which is implemented by a number of different physical processor designs. Each implementation can make different trade-offs in terms of size and performance but still offers the same basic interface to the programmer: the processors still execute the same programs and yield the same results, they just do so in a different way.

Regardless of the actual implementation, the core operations carried out by a stored program computer are easy to describe. The one assumption is that the processor maintains a **Program Counter** (*PC*), or **instruction pointer**, that determines where in memory the instruction which should be executed next is; normally we expect the *PC* to be set to zero when the processor is reset. Once the processor is started, execution of the program proceeds using a slight alteration to our original scheme:

1. Encode a program onto a tape; load this tape into memory using the tape reader, zero *PC* and start the computer.
2. From the address in *PC*, fetch the next instruction *IR*.
3. Increment *PC* so it points to the next instruction.
4. If *IR* is the *HALT* instruction then halt the computer, otherwise execute the instruction *IR* (i.e., perform the operation it specifies).
5. Repeat from step 2.

More formally, once execution starts this repeated sequence of steps encapsulates what is normally called the **fetch-decode-execute cycle**:

Fetch During the first stage, the control unit uses the program counter to access memory and load the instruction to be executed next. The program counter is then incremented, using the ALU, so that it points to the next instruction in the program.

Decode Once we have retrieved the instruction, we need to check what it means in terms of the operations it instructs the processor to perform. From the numerical instruction the control unit extracts the operation code, or opcode, that specifies the computation we perform and the operands that we use as values in that computation. The operands can take several forms: they can refer to an accumulator or a memory location, whose content we must fetch, or simply an immediate number whose value is directly available.

Execute At this point, the processor has all the information required to actually do something; we say the instruction is now executed. During execution the control unit might invoke the ALU to perform some computation, or access memory to load or store values into or from an accumulator. Any results generated by these actions are potentially written back into an accumulator or into memory for use in the execution of further instructions.

To complete our imaginary computer, consider an "upgrade" so that it uses a stored program architecture rather than a Harvard architecture. All we need to unify data and instructions is an encoding (or map) from values to meaning; essentially we have to define the ISA for the computer. For our purposes, the encoding does not matter too much; imagine that we again represent everything as six-digit decimal values. An example encoding, where we can fill in n as a value of our choosing, might be as follows:

- $0nnnnn$ means $A \leftarrow n$.
- $1nnnnn$ means $MEM[n] \leftarrow A$.
- $2nnnnn$ means $A \leftarrow MEM[n]$.
- $3nnnnn$ means $A \leftarrow A + MEM[n]$.
- $4nnnnn$ means $A \leftarrow A - MEM[n]$.
- $5nnnnn$ means $HALT$.

For example, the value 300020 is viewed as the instruction $A \leftarrow A + MEM[20]$. In this context the "3" part is the opcode and the "00020" part is the operand.

Equipped with our upgraded computer and a definition of the ISA, we can look at how the example program from above would be executed. Notice that the program is the same four instructions

$$\begin{array}{ll} A \leftarrow MEM[4] & \mapsto 200004 \\ A \leftarrow A + MEM[5] & \mapsto 300005 \\ MEM[6] \leftarrow A & \mapsto 100006 \\ HALT & \mapsto 500000 \end{array}$$

CPU		
state	=	start
PC	=	0
IR	=	
	=	
A	=	000000

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_0 starts by initialising the processor and loading the program into memory.

CPU		
state	=	fetch
PC	=	0
IR	=	200004
	=	$A \leftarrow MEM[4]$
A	=	000000

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_1 fetches instruction #0 from memory address $PC = 0$.

CPU		
state	=	execute
PC	=	1
IR	=	200004
	=	$A \leftarrow MEM[4]$
A	=	000010

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_2 increments *PC* and executes instruction #0, loading the value 10 from memory address 4 into *A*.

CPU		
state	=	fetch
PC	=	1
IR	=	300005
	=	$A \leftarrow A + MEM[5]$
A	=	000010

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_3 fetches instruction #1 from memory address $PC = 1$.

CPU		
state	=	execute
PC	=	2
IR	=	300005
	=	$A \leftarrow A + MEM[5]$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_4 increments *PC* and executes instruction #1, adding the value 20 from memory address 5 to *A*.

Figure 4.8 Execution of a simple program on a stored program computer.

CPU		
state	=	fetch
PC	=	2
IR	=	100006
	=	$MEM[6] \leftarrow A$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_5 fetches instruction #2 from memory address $PC = 2$.

CPU		
state	=	execute
PC	=	3
IR	=	100006
	=	$MEM[6] \leftarrow A$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000030	
7	000000	

Step t_6 increments *PC* and executes instruction #2, storing the value 30 in memory address 6.

CPU		
state	=	fetch
PC	=	3
IR	=	500000
	=	*HALT*
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000030	
7	000000	

Step t_7 fetches instruction #3 from memory address $PC = 3$.

CPU		
state	=	execute
PC	=	4
IR	=	500000
	=	*HALT*
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[6] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000030	
7	000000	

Step t_8 increments *PC* and executes instruction #3, halting the computer.

Figure 4.9 Execution of a simple program on a stored program computer.

except now we map them, via our instruction encoding, into their representation as six-digit decimal values. The steps involved in executing the program (and the memory content at each step) are detailed in Figure 4.8 and Figure 4.9. Two key differences are obvious:

1. we have removed the need for a tape *during execution* since the program is stored in memory at addresses $0 \ldots 3$, and
2. the CPU now has a program counter (which is analagous to the tape head we needed before) that keeps track of where in memory the next instruction comes from.

Other than that, execution is similar and after eight steps, we reassuringly end up with the result stored in $MEM[6]$.

4.5 Toward Modern Computers

To reiterate, two main features can be used to distinguish a stored program architecture from previous designs: the inclusion of a memory that stores *both* instructions and data, and the segregation of memory and control unit. These features were used to good effect by two of the first electronic stored program computers, the **Small-Scale Experimental Machine (SSEM)** and **Electronic Delay Storage Automatic Calculator (EDSAC)**.

The SSEM [36], nicknamed the Baby, was designed and built by a team led by Frederic Williams and Tom Kilburn at the University of Manchester. It is generally recognised as the first computer to run a stored program; in 1948 it used a program for division to show that the numbers 314159265 and 217828183 are relatively prime. The architecture of the SSEM was based around a device called the Williams tube that acted as a memory. The device was able to store 32 values, each 32 bits in size. Values were loaded from the memory into an accumulator and then use as input one of a limited set of arithmetic operations with results stored back into the memory; this represents a typical cycle in stored program architectures.

Like the SSEM, the EDSAC [11, 66] was architected around a memory technology, in this case a device called a mercury delay tube that was designed to hold 1024 values each of 18 bits in size; it eventually had only 512 values each 17 bits in size . The EDSAC was constructed by Maurice Wilkes [67] and his team at the University of Cambridge with the support of J. Lyons & Co. It ran a stored program slightly later than the SSEM; in 1949 it ran a program that calculated a list of prime numbers. Even so, it offered a number of other firsts. David Wheeler, a researcher on the EDSAC project, was credited with the invention of what we now know as the subroutine. In addition, the EDSAC was equipped with the first example of what we now know as an operating system. The so-called **Initial Orders** [65] represented a short program initialised by a plug-board that were responsible for loading the user program from a storage device. Because of these innovations the EDSAC is

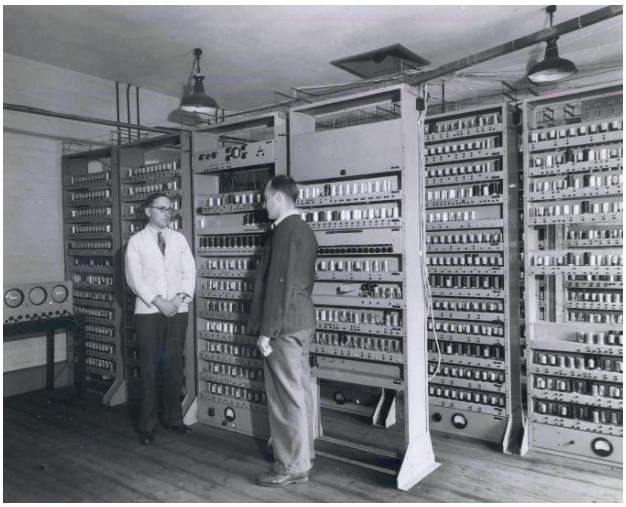


Figure 4.10 The EDSAC computer, including William Renwick and Maurice Wilkes. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/edsac_wilkes.jpg`)

Figure 4.11 Sid Barton with 9-inch tubes which implemented the EDSAC operator display. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/edsac99.35.jpg`)

generally recognised as the first practical stored program computer and the first real forerunner of computers as we know them today.

As implementations of the general stored program architecture, the SSEM and EDSAC both presented ways to write practical and sizable programs for the first time. Several additional aspects of the architecture, which bridge the gap somewhat toward more modern computers, are worth discussing.

4.5.1 The von Neumann Bottleneck

The execution of the fetch-decode-execute cycle highlights an issue with the basic stored program architecture: processor performance is tightly bound to the memory performance. That is, since we need to access memory at least once per cycle to retrieve an instruction, the processor can only operate as fast as the memory. This is sometimes termed the **von Neumann bottleneck** or **memory wall** [68] and presents a significant problem for modern computer processors.

We investigate the construction of memory in Chapter 8 but roughly speaking the memory wall is exacerbated by two main facts: firstly, since the memory is physically separated from the processor, there is an inherent and significant delay in transferring instructions or data; secondly, the development of memory technologies at a physical level has been outstripped by improvements to processor technology. This is partly because their goals differ (e.g., memory technology is often concerned with size rather than speed), but the end result is that a processor can commonly be starved of instructions to execute due to the latency of memory access. Approaches to removing, or at least hiding, this latency are therefore enormously important.

4.5.2 Data-Dependent Control-Flow

The addition of the program counter has a fairly profound implication for the types of program we can write. Thus far, we have examined a (very simple) straight-line example program: execution starts at the first instruction and proceeds with each subsequent instruction in sequence until the CPU is halted. This behaviour is enforced by the fetch-decode-execute cycle which increments *PC* to point at the next instruction in the sequence. However, imagine we add some instructions to our ISA which allow the program to test *A* and (conditionally) set the value of *PC*. For example, imagine we add three instructions:

- 6*nnnnn* means $PC \leftarrow n$, i.e., set the value of *PC* so that the next instruction is fetched from address *n* in memory.
- 7*nnnnn* means if $A = 0, PC \leftarrow n$, i.e., set the value of *PC* to *n* if the accumulator *A* holds the value 0, otherwise do not change it at all.
- 8*nnnnn* means if $A \neq 0, PC \leftarrow n$, i.e., set the value of *PC* to *n* if the accumulator *A* holds the value other than 0, otherwise do not change it at all.

Using these instructions, the program can *dictate* which instruction the processor executes next by controlling the value of *PC*. Furthermore, it can do this based on the result of some previous computation (i.e., based on the value held in *A*). We say this allows the program to alter the **control-flow**, or the path taken through the program, in a way which is data-dependent.

Imagine we take our example program to add together numbers and replace the *HALT* instruction with one that alters *PC*. We can construct what you might recognise as an infinite loop: once the program finishes the addition and stores the result into $MEM[6]$, it sets $PC = 0$:

$$\begin{aligned} &A \leftarrow MEM[4] \\ &A \leftarrow A + MEM[5] \\ &MEM[6] \leftarrow A \\ &PC \leftarrow 0 \end{aligned}$$

This means that the instruction executed after instruction #3 will be instruction #0, i.e., the whole process of addition is repeated again and again until the computer is turned off ! Clearly by more intelligent use of these sorts of instruction one can construct programs that include the equivalent of iteration (e.g., `for` or `while`) and conditional (e.g., `if`) statements which one might associate with C programs.

4.5.3 Self-Modifying Programs

There is a subtle difference between Harvard and stored program architectures in that with the latter design, instructions and data are the same thing so we can alter instructions just as easily as data: in the former case we could not alter the program once execution had started. Imagine we make a mistake while writing our example program so that instead of storing the addition result in $MEM[6]$, we store it in $MEM[3]$:

$$\begin{aligned} &A \leftarrow MEM[4] \\ &A \leftarrow A + MEM[5] \\ &MEM[3] \leftarrow A \\ &HALT \end{aligned}$$

Examining the execution in Figure 4.12 and Figure 4.13 we see that when instruction #2 is executed, it actually overwrites instruction #3 and thus changes it from *HALT* to $A \leftarrow 30$. As a result, when instruction #3 is fetched and executed it reloads the value 30 into *A* and then carries on rather than halting execution: the program has modified itself by altering instructions while the same program is running !

At face value, this seems like an issue of program correctness: if we make a mistake in writing the program, it should not be a surprise that it executes in an erroneous manner. However, there are situations where the constructive use of self-modifying programs allows one to consider novel solutions to problems; we examine two such situations below.

CPU		
state	=	start
PC	=	0
IR	=	
	=	
A	=	000000

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_0 starts by initialising the processor and loading the program into memory.

CPU		
state	=	fetch
PC	=	0
IR	=	200004
	=	$A \leftarrow MEM[4]$
A	=	000000

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_1 fetches instruction #0 from memory address $PC = 0$.

CPU		
state	=	execute
PC	=	1
IR	=	200004
	=	$A \leftarrow MEM[4]$
A	=	000010

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_2 increments *PC* and executes instruction #0, loading the value 10 from memory address 4 into *A*.

CPU		
state	=	fetch
PC	=	1
IR	=	300005
	=	$A \leftarrow A + MEM[5]$
A	=	000010

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_3 fetches instruction #1 from memory address $PC = 1$.

CPU		
state	=	execute
PC	=	2
IR	=	300005
	=	$A \leftarrow A + MEM[5]$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_4 increments *PC* and executes instruction #1, adding the value 20 from memory address 5 to *A*.

Figure 4.12 Execution of a self-modifying program on a stored program computer.

CPU		
state	=	fetch
PC	=	2
IR	=	100003
	=	$MEM[3] \leftarrow A$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	500000	*HALT*
4	000010	
5	000020	
6	000000	
7	000000	

Step t_5 fetches instruction #2 from memory address $PC = 2$.

CPU		
state	=	execute
PC	=	3
IR	=	100003
	=	$MEM[3] \leftarrow A$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	000030	$A \leftarrow 30$
4	000010	
5	000020	
6	000000	
7	000000	

Step t_6 increments *PC* and executes instruction #2, storing the value 30 in memory address 3 and thereby overwriting the *HALT*.

CPU		
state	=	fetch
PC	=	3
IR	=	000030
	=	$A \leftarrow 30$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	000030	$A \leftarrow 30$
4	000010	
5	000020	
6	000000	
7	000000	

Step t_7 fetches the modified instruction #3 from memory address $PC = 3$.

CPU		
state	=	execute
PC	=	4
IR	=	000030
	=	$A \leftarrow 30$
A	=	000030

MEM		
Address	*Value*	*Meaning*
0	200004	$A \leftarrow MEM[4]$
1	300005	$A \leftarrow A + MEM[5]$
2	100006	$MEM[3] \leftarrow A$
3	000030	$A \leftarrow 30$
4	000010	
5	000020	
6	000000	
7	000000	

Step t_8 increments *PC* and executes the modified instruction #3, loading the value 30 into *A*.

Figure 4.13 Execution of a self-modifying program on a stored program computer.

4.5.3.1 Program Relocation

Considering the original example program again, notice that it only works if we load the instructions into memory starting at address 0. For example, imagine if we load it starting at address 1 so that the initial memory content is:

MEM		
Address	*Value*	*Meaning*
0		
1	200004	$A \leftarrow MEM[4]$
2	300005	$A \leftarrow A + MEM[5]$
3	100006	$MEM[6] \leftarrow A$
4	500000	*HALT*
5	000010	
6	000020	
7	000000	

We can expect at least two problems when executing this program. Firstly, we do not really know what the contents of $MEM[0]$ will be, so if we start executing at address 0 undefined behaviour will result. Secondly, our load and store instructions have fixed addresses in them; when we try to load 000010 from address 4, we actually load the encoded *HALT* instruction instead, i.e., the value 500000. To resolve this problem, we need to **relocate** instructions by patching addresses so they make sense relative to the address we load the program at.

The Initial Orders operating system used by EDSAC had a mechanism to do just this. It maintained a value called Θ which held the **load point**, i.e., the address where the program is loaded. As each instruction was read into memory, the address field n was modified by adding Θ to it. That is, an instruction $A \leftarrow MEM[4]$ would be modified to $A \leftarrow MEM[4+\Theta]$. This solves the second of our problems above; setting $\Theta = 1$ the loaded program becomes

MEM		
Address	*Value*	*Meaning*
0		
1	200005	$A \leftarrow MEM[4+1]$
2	300006	$A \leftarrow A + MEM[5+1]$
3	100007	$MEM[6+1] \leftarrow A$
4	500000	*HALT*
5	000010	
6	000020	
7	000000	

such that, for example, the load instruction now correctly loads the value 000010 from address $4+1=5$ rather than from address 4.

4.5.3.2 Array Iteration

Another use of self-modifying code is to cope with inadequacies in the ISA, for example the problem that we must supply addresses as constant values. Imagine a simple problem involving a 3-element vector X where we would like to write a program that loads X_i, increments it and then stores the result back in the same place. In C this would be simple: we would represent X as an array `X` and write a loop such as

```
for( int i = 0; i < 3; i++ ) {
  X[ i ] = X[ i ] + 1;
}
```

such that to access element `X[i]` we do some pointer arithmetic and load from or store to address `&X+i`; in this context `&X` is the base address and `i` is the offset. But this is impossible with the instructions at our disposal; the address must be specified as *part* of the instruction, not *calculated* as one would in C. As a result, to implement this sort of loop on a computer such as EDSAC, one was forced to employ self-modifying code in a constructive way. The basic approach, focusing on the instruction that loads the value of $X[i]$, is that we start by saying the instruction will load $X[0]$. After executing it, we modify the same instruction so that next time around the loop it will load $X[1]$ and so on. Consider a short example of this in action where only the instructions that represent the loop body are shown:

MEM		
Address	*Value*	*Meaning*
0	200015	$A \leftarrow MEM[15]$
1	300014	$A \leftarrow A + MEM[14]$
2	100015	$MEM[15] \leftarrow A$
3	200000	$A \leftarrow MEM[0]$
4	300014	$A \leftarrow A + MEM[14]$
5	100000	$MEM[0] \leftarrow A$
6	200002	$A \leftarrow MEM[2]$
7	300014	$A \leftarrow A + MEM[14]$
8	100002	$MEM[2] \leftarrow A$
.	.	.
.	.	.
.	.	.
14	000001	
15	$X[0]$	
16	$X[1]$	
17	$X[2]$	

Notice that $MEM[14]$ contains the constant 1 and that the values $X[0\ldots2]$ are held in $MEM[15\ldots17]$. Instructions #0…#2 load the value $X[0]$ from $MEM[15]$, increment the value by adding the value at $MEM[14]$, i.e., the constant 1, and then store $X[0]+1$ back into $MEM[15]$. This completes the first iteration of the loop. But before we branch back to execute the second iteration, we execute instructions #3…#8 which act to self-modify the program. For example, instructions #3…#5 load the value from $MEM[0]$, i.e., the instruction $A \leftarrow MEM[15]$ represented by value 200015, increment the value by adding the contents of $MEM[14]$, and then store the resulting value 200016 back into $MEM[0]$. This has the net effect of modifying $A \leftarrow MEM[15]$ into $A \leftarrow MEM[16]$ so that in the next loop iteration, we load and increment $X[1]$. Instructions #6…#8 perform a similar task for the store instruction in $MEM[2]$.

Clearly this is quite an involved process for writing a loop we now regard, at a high-level at least, as fairly simple; the complexity and program length are basically due to the limited form of our ISA. To improve matters, we will see later that one can allow different types or modes of load and store instruction. In the context of our CPU for example, we might introduce a second accumulator called B and then redefine load and store instructions so that B is added to the address n. For example, what was $A \leftarrow MEM[n]$ would become $A \leftarrow MEM[B+n]$. By allowing the pro-

grammer to set the value of B, we have constructed what might be called an indexed mode for loads and stores.

4.6 Further Reading

- P.E. Ceruzzi.
  A History of Modern Computing.
  MIT Press, 2003. ISBN: 0-262-53203-4.
- G. O'Regan.
  A Brief History of Computing.
  Springer-Verlag, 2008. ISBN: 1-848-00083-9.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.
- B. Randell.
  The Origins of Digital Computers: Selected Papers.
  Springer-Verlag, 1982. ISBN: 0-387-11319-3.

4.7 Example Questions

18. The description of a stored program memory architecture often implies it is superior to, or at least more advanced than, a Harvard memory architecture. In reality both have advantages and disadvantages; discuss **two** advantages that a Harvard memory architecture can provide.

19. Consider a simple stored program computer with an instruction set of nine instructions whose semantics and encoding are defined by:

- $0nnnnn$ means $A \leftarrow n$.
- $1nnnnn$ means $MEM[n] \leftarrow A$.
- $2nnnnn$ means $A \leftarrow MEM[n]$.
- $3nnnnn$ means $A \leftarrow A + MEM[n]$.
- $4nnnnn$ means $A \leftarrow A - MEM[n]$.
- $5nnnnn$ means $HALT$.
- $6nnnnn$ means $PC \leftarrow n$.
- $7nnnnn$ means if $A = 0, PC \leftarrow n$.
- $8nnnnn$ means if $A \neq 0, PC \leftarrow n$.

Making appropriate assumptions as required, write a program which will emulate the behaviour (i.e., perform the fetch-decode-execute cycle) after a program is loaded into memory and the computer is reset.

20. Using your emulator from the previous question, write and describe the operation of a program (i.e., specify the initial memory content) to compute the sum of n elements held in an array X. That is, compute

$$\sum_{i=0}^{n-1} X_i$$

for some known n where X is located at some known address m.

Chapter 5
Basic Processor Design

People who are really serious about software should make their own hardware.

– A. Kay

Abstract In previous chapters we introduced the physical characteristics of digital circuits and the mechanisms used to design and implement them, followed by a functional description of how a computer processor executes programs. We can now combine this experience by examining how to design and implement a concrete processor. In this chapter we focus on the high-level design issues rather than the design of individual components. As such, we cover micro-architecture design (the design of the physical hardware) and instruction set design (the design of the interface to the hardware offered to the programmer). In other books, it is popular to introduce assembly language at this point as a means of programming the resulting system. We take a different approach, opting to describe instructions (and hence programs) using the pseudo-assembly language introduced previously.

5.1 A Concrete Stored Program Architecture

To investigate some of the issues involved in designing a concrete instruction set and implementing a concrete processor, we will step through the development of a realisation of MIPS32 [42–44]. **Micro-Processor without Interlocked Pipeline Stages (MIPS)** is a name that relates to an implementation technique called pipelining which we will discuss in Chapter 9. The first MIPS processor that used the associated techniques was developed in the early 1980s by a team led by John Hennessy at Stanford University. Their focus was improvement of performance using pipelining techniques but, recalling that an ISA is simply an interface not a design, the MIPS32 ISA does not dictate this as necessary. This is highlighted by the fact that the company formed to commercialise the Stanford design, also called MIPS, has produced a series of backwardly-compatible MIPS32 ISAs which are now used in implementations by companies such as NEC, Toshiba and Phillips; their products

Instruction	Meaning	Encoding
`add`	$GPR[rd] \leftarrow (GPR[rs] + GPR[rt])$	`|000000|..rs.|..rt.|..rd.|00000|100000|`
`sub`	$GPR[rd] \leftarrow (GPR[rs] - GPR[rt])$	`|000000|..rs.|..rt.|..rd.|00000|100010|`
`mul`	$GPR[rd] \leftarrow (GPR[rs] \cdot GPR[rt])$	`|011100|..rs.|..rt.|..rd.|00000|000010|`
`mult`	$(HI, LO) \leftarrow (GPR[rs] \cdot GPR[rt])$	`|000000|..rs.|..rt.|00000|00000|011000|`
`and`	$GPR[rd] \leftarrow (GPR[rs] \wedge GPR[rt])$	`|000000|..rs.|..rt.|..rd.|00000|100100|`
`or`	$GPR[rd] \leftarrow (GPR[rs] \vee GPR[rt])$	`|000000|..rs.|..rt.|..rd.|00000|100101|`
`xor`	$GPR[rd] \leftarrow (GPR[rs] \oplus GPR[rt])$	`|000000|..rs.|..rt.|..rd.|00000|100110|`
`nor`	$GPR[rd] \leftarrow (GPR[rs] \bar{\vee} GPR[rt])$	`|000000|..rs.|..rt.|..rd.|00000|100111|`
`sllv`	$GPR[rd] \leftarrow (GPR[rs] \ll GPR[rt]_{4\ldots0})$	`|000000|..rs.|..rt.|..rd.|00000|000100|`
`srlv`	$GPR[rd] \leftarrow (GPR[rs] \gg GPR[rt]_{4\ldots0})$	`|000000|..rs.|..rt.|..rd.|00000|000110|`
`addi`	$GPR[rt] \leftarrow (GPR[rs] + ext(imm))$	`|001000|..rs.|..rt.|.............imm.|`
`mfhi`	$GPR[rd] = HI$	`|000000|00000|00000|..rd.|00000|010000|`
`mflo`	$GPR[rd] = LO$	`|000000|00000|00000|..rd.|00000|010010|`
`mfc0`	$GPR[rt] = CP0[rd]$	`|010000|00000|..rt.|..rd.|00000|000000|`
`mthi`	$HI = GPR[rs]$	`|000000|..rs.|00000|00000|00000|010001|`
`mtlo`	$LO = GPR[rs]$	`|000000|..rs.|00000|00000|00000|010011|`
`mtc0`	$CP0[rd] = GPR[rt]$	`|010000|00100|..rt.|..rd.|00000|000000|`
`lw`	$GPR[rt] \leftarrow MEM[GPR[rs] + ext(imm)]$	`|100011|..rs.|..rt.|.............imm.|`
`sw`	$MEM[GPR[rs] + ext(imm)] \leftarrow GPR[rt]$	`|101011|..rs.|..rt.|.............imm.|`
`beq`	if $GPR[rs] = GPR[rt], PC \leftarrow PC + ext(off : 00_{(2)})$	`|000100|..rs.|..rt.|.............off.|`
`bne`	if $GPR[rs] \neq GPR[rt], PC \leftarrow PC + ext(off : 00_{(2)})$	`|000101|..rs.|..rt.|.............off.|`
`blez`	if $GPR[rs] \leq 0, PC \leftarrow PC + ext(off : 00_{(2)})$	`|000110|..rs.|00000|.............off.|`
`bgtz`	if $GPR[rs] > 0, PC \leftarrow PC + ext(off : 00_{(2)})$	`|000111|..rs.|00000|.............off.|`
`bltz`	if $GPR[rs] < 0, PC \leftarrow PC + ext(off : 00_{(2)})$	`|000001|..rs.|00000|.............off.|`
`bgez`	if $GPR[rs] \geq 0, PC \leftarrow PC + ext(off : 00_{(2)})$	`|000001|..rs.|00001|.............off.|`
`b`	$PC \leftarrow PC + ext(off : 00_{(2)})$	`|000100|00000|00000|.............off.|`
`j`	$PC \leftarrow (imm : 00_{(2)})$	`|000010|.........................imm.|`
`jr`	$PC \leftarrow GPR[rs]$	`|000000|..rs.|00000|00000|00000|001000|`
`jal`	$GPR[31] \leftarrow PC + 8, PC \leftarrow (imm : 00_{(2)})$	`|000011|.........................imm.|`
`nop`	no operation	`|000000|00000|00000|00000|00000|000000|`
`syscall`	cause system call exception	`|000000|00000000000000000000|001100|`
`eret`	return from exception	`|010000|10000000000000000000|011000|`

Table 5.1 A tabular description of a reduced MIPS32 ISA.

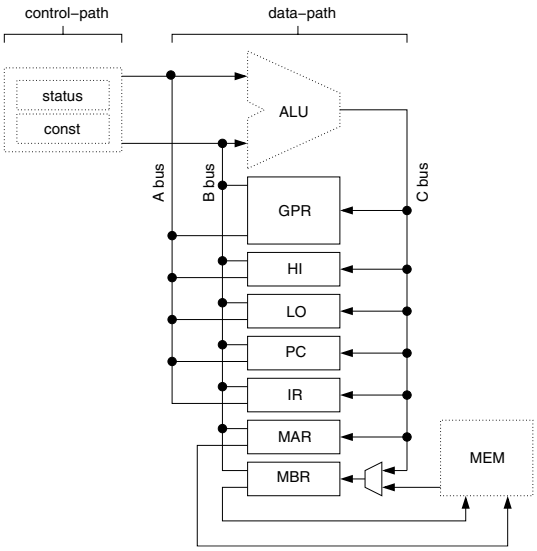


Figure 5.1 An simple data-path that could be used to implement the MIPS32 ISA.

drive consumer electronics such as the PlayStation2 and workstations such as the SGI Indy.

We selected MIPS32 as an example for several reasons. The design is fairly simple and very typical of modern processors; this is highlighted by the level of real-world use of the ISA. The documentation for the ISA is easily available and there are a number of good compilers, simulators and so on that can be used to explore the design without physical hardware. Unlike more complete references [28], we focus on a reduced version of the MIPS32 ISA to make it more easily accessible; this is shown in Table 5.1. For example, we omit all floating point operations. Where appropriate we are careful to differentiate between the full MIPS32 ISA and this reduced version.

5.1.1 Major Data-path Components

We often call the details of a processor implementation the **micro-architecture** to contrast with the **instruction set architecture** which represents an interface with the programmer potentially shared between many different implementations. The MIPS32 ISA dictates, or at least strongly hints toward a 32-bit micro-architecture: the data-path width is 32 bits in the sense that the accumulators hold 32-bit values, the buses usually transfer 32-bit values and the ALU can usually deal with 32-bit

inputs. In some sense, 32-bit is the natural size of things within the processor. To highlight where this size has an impact we say the data-path is n bits wide: keep in mind that in this case $n = 32$. Figure 5.1 shows a block diagram of a data-path which we use for our implementation. The components in, and structure of, the diagram closely follows the description of the more general stored program architecture. Sectioned to relate the concrete data-path to the abstract stored program architecture, the key components are:

Accumulators The accumulators within a stored program architecture are defined in a somewhat abstract way. Our data-path includes a set of concrete accumulators, normally called **registers** in a modern context, which are for general-purpose use. There are thirty two such registers grouped into a structure called a **register file**; we write $GPR[i]$ to refer to the i-th general-purpose register, each value in $GPR[i]$ is n bits in size. The register file is like a small, high-speed, short-term memory whose content we can access very quickly making it ideal to supply values to the ALU for computation and storing results of said computation before they are written into more long-term storage such as the main memory. The registers are general-purpose in the sense that their content and usage are freely determined by the programmer. Usually the general purpose registers are used to contain integer values; there may be a second register file to store floating point values.

The MIPS32 ISA demands that $GPR[0]$ is the constant value zero; retrieving the value of $GPR[0]$ gives zero, any stores to $GPR[0]$ are discarded. This might appear a strange choice: why would we waste a register to store a seemingly useless value and nothing else ? The answer is that by having the constant zero always available we can construct some operations within the ISA from others using no additional logic. For example consider the `nop` or NOP instruction which performs no operation. We can implement such an instruction with more or less any instruction which targets $GPR[0]$ and has no other side effects. For example, using the instruction `sllv` to shift $GPR[0]$ left by a distance of zero and store the result in $GPR[0]$ will have no effect on the state of the processor: although we need some logic to implement the `sllv` instruction, we get the `nop` instruction for free.

As well as the general-purpose registers, the data-path includes a number of special-purpose registers whose usage is pre-determined by the design of the processor; these might not even be accessible to the programmer. The special-purpose register set includes a **Program Counter** (*PC*) that holds the address of the next instruction the processor will execute; an **Instruction Register** (*IR*) which holds the instruction currently being executed; the **Memory Address Register** (*MAR*) which holds addresses that are passed to the memory when a value is stored or retrieved; the **Memory Buffer Register** (*MBR*) which holds the actual values being stored or retrieved; and the *HI* and *LO* registers that act as temporary storage for specific instructions such as `mult`.

The issues of designing and constructing registers and register files are examined in more detail in Chapter 8.

Arithmetic and Logic Unit (ALU) The ALU is shown as a dashed wedge shape. We will discuss implementation of devices that perform arithmetic and logic in Chapter 7; for now it suffices to understand that the ALU simply performs computation for us. More specifically, where we might write a Verilog statement to perform an addition we would instead have the ALU produce a result by passing it operands and using an opcode to select the addition operation.

Control Unit The control unit is shown alongside the data-path in a dashed box; it has implicit links into all the components so that it can control how they behave. Inside the control unit are two implicit structures whose implementation we leave as fairly abstract. Firstly, the control unit has a constant value register which it uses to send constant values onto either the *A* or *B* buses; in reality this might not be a physical register. Secondly, the control unit has a status register which details the state of the processor. Various conditions that are flagged by bits in a typical status register include arithmetic exceptions, for example division by zero, and the processor mode (either user or kernel), an issue which is discussed later in Chapter 12. A program can query the status register by extracting and examining bits and hence make decisions on how to proceed based on special events which might have occurred during execution.

Buses There are two main groups of buses within the data-path which are used to communicate values between the different components: the ALU buses allow communication of operands and results to and from the ALU, the memory bus allows communication of addresses and values to and from the memory.

There is one slight subtlety in the design of our buses in that where a value can be communicated from two different places we need a multiplexer to select which one; control signals for the multiplexer are managed by the control unit. Examples include the input to the *MBR* which can take values from either the *C* bus or the memory.

Input and Output Devices In this basic model we largely ignore input and output devices and assume they interact with the processor. Simply put, this involves associating regions of memory with a hardware device rather than a memory location. So for example if one were to write into address 123 this might actually mean "communicate a value to the video device" rather than "store a value in memory".

Memory The main memory is shown alongside the data-path as a dashed box since it is not actually part of the processor; memory is commonly placed on a different physical microchip to the processor. From here on we will consider a memory in which each location holds 8-bit values. Each location will have an address so this is called a byte addressed memory: we are able to access each byte individually. However, in order to remove the need for multiple accesses to retrieve or store a 32-bit value for example, the basic unit of transfer between the processor and memory typically matches the data-path width n. This does not change the fact that the memory is byte addressed however.

We still view the memory fairly abstractly here, filling in the details of implementation in Chapter 8. For now it suffices to understand that the memory can be represented, if not actually implemented, as a 2-dimensional Verilog register.

The crucial thing to realise from this description is that the data-path contains *only* hardware devices we already know how to build: we know how to implement registers in Verilog; we know how to implement simple arithmetic circuits in Verilog; we know how to design and implement state machines in Verilog. Although we will use later chapters to go into more detail as to how these devices are actually realised, there should be nothing intimidating about the block diagram in Figure 5.1: it is just a big circuit.

5.1.2 Describing Instruction Behaviour

In other books, it is popular to introduce assembly language early on as a means of programming the processor under discussion. We break with this trend and take a slightly different approach. That is, we describe instructions (and hence programs) using a pseudo-assembly language that relates directly to the effect of each instruction on the processor. The aim is to show that machine code is the natural language at this level: only when we start to discuss the interface between hardware and software in Chapter 10 and Chapter 11 is assembly language introduced.

The general-purpose registers are viewed abstractly as an array such that use of an index identifies a specific register, $GPR[7]$ is the general-purpose register number seven for example. Any special-purpose registers are referred to by their name, for example *PC* is the program counter. Memory is also viewed abstractly as an array. Recalling Table 5.1, an instruction with the meaning

$$GPR[2] \leftarrow GPR[3] + GPR[4]$$

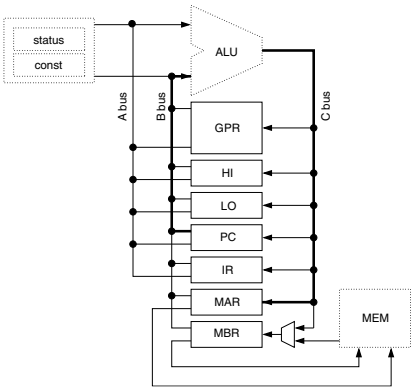
shows the use of operands read from general-purpose register three and four in performing an ALU operation (in this case an addition), and writing a result back into general-purpose register two. Roughly speaking, things on the right-hand side of the assignment symbol $\leftarrow$ should be interpreted as being read (or as source operands) and those on the left-hand side as being written (or as target operands) by the instruction. This also implies that memory access can be written in the same way; for example the operation

$$MEM[64] \leftarrow GPR[2]$$

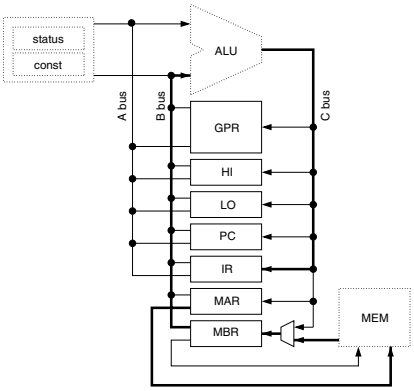
reads general-purpose register two and writes the value into memory address sixty four.

Where an instruction is only executed if some condition is true, we adopt a pseudo-code style and write the condition before the instruction. For example, in writing the instruction

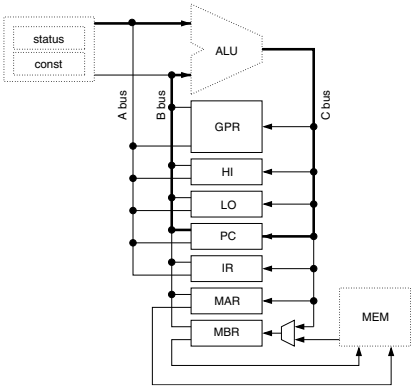
$$\text{if } GPR[2] = 0, PC \leftarrow PC + 12$$



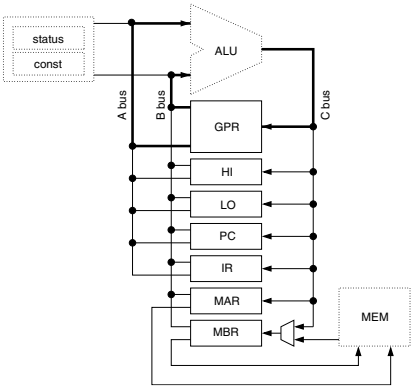
(a) Step t_0 passes *PC* through ALU to *MAR*.



(b) Step t_1 loads instruction, at address in *MAR*, from memory and placed in *IR*.



(c) Step t_2 increments *PC*, using the ALU, to point at next instruction.



(d) Step t_3 executes instruction in *IR* by reading operands from *GPR*, using ALU to perform addition and then writing result back into *GPR*.

Figure 5.2 Execution of an addition instruction on the example data-path.

we mean that the *PC* is only incremented by twelve if the related condition is satisfied. That is, if the contents of general-purpose register two is equal to zero, then the current *PC* value is incremented, otherwise no action is performed.

5.1.3 The Fetch-Decode-Execute Cycle

To see how the data-path is typically used to execute instructions, consider the example of $GPR[2] \leftarrow GPR[3] + GPR[4]$. Figure 5.2 details the familiar steps from Chapter 4 in a graphical way; they are also described below:

Fetch The *PC* is fed through the ALU unaltered and stored in the *MAR* since there is no direct connection between *PC* and *MAR*. Once stored, the control unit initiates a load from memory; this results in the instruction at the address in *PC* being loaded into the *MBR*. The instruction is again fed through the ALU and stored in the instruction register *IR*. Finally, the *PC* is incremented by four so it points at the next instruction to be executed: the *PC* is fed onto the *A* bus while the constant 4 is fed onto the *B* bus, the ALU is invoked to perform an addition with the result on the *C* bus stored back in the *PC*. Note that we increment the *PC* by four because each instruction is 32 bits in size and our *PC* counts in bytes: a 32-bit instruction is four 8-bit bytes.

Decode The control unit now examines the instruction in *IR* and determines the opcode and operands: in this case the opcode tells us we should do an addition and the operands tell us to source values from $GPR[3]$ $GPR[4]$ and produce a result in $GPR[2]$.

Execute Execution proceeds by placing $GPR[3]$ onto bus *A* and $GPR[4]$ onto bus *B*, then provoking the ALU to add the two inputs together; the result is produced on bus *C*.

Write Finally, the write phase takes the value on bus *C* and stores it into $GPR[2]$. Execution of this instruction is now complete and the processor loops back to the fetch stage to retrieve and execute the next instruction.

The actions within the execute and write stages will change depending on the instruction type. For example, if we are required to perform a load from memory, then it is possible the ALU will not be needed whereas if we are performing an arithmetic operation it will. The actions within the fetch and decode stages are the same for every instruction. This should make sense since until the instruction decode has taken place, the control unit does not even know what the instruction means.

5.1.4 Controlling the Data-path

Construction of the control unit, which constitutes the control-path of the processor, is crucial: if the control unit cannot efficiently manage the resources in the data-path to implement the fetch-decode-execute cycle, the speed of the entire processor is compromised even if those resources are individually capable of being more efficient. There are two main approaches to designing the physical logic within the control unit; these are commonly termed hard-coded and micro-code designs.

In a hard-coded design one uses techniques outlined in Chapter 3 to directly implement a state machine capable of operating the data-path components in the right

way. This technique is easy to implement, although the state machine can become complicated, and is efficient in the sense that the amount of logic used can be small since the state machine is tailored closely to what is required. However, once the state machine is implemented in logic it is fixed forever: altering it means building a new processor.

A **micro-coded** design approaches the same problem in a different way. It essentially allows the data-path to be programmed using very simple instructions called micro-instructions; an instruction in the ISA is therefore implemented by a number of micro-instructions. One *must* contrast the level at which these two types of instruction work: an instruction in the ISA operates at a fairly high-level, the operands involved are typically values from the register set; an instruction in the micro-code operates at a low-level, typical operands are control signals in the data-path. A micro-coded design is more complex to implement but offers some significant advantages in terms of flexibility. For example, one can fix problems with the processor by upgrading the micro-code rather than re-engineering the actual hardware. This exact issue reared its head in 1994 when the Intel Pentium range of processors was found to have a bug in the division instruction; when using certain inputs the instruction would produce the wrong result. Today, Intel processors allow upgradeable micro-code which could potentially have solved this problem without the associated cost of product recall: in theory they might have simply issued software that would have updated the micro-code and fixed the flaw.

5.2 Buses

As previously stated, a **bus** is a communication link between devices; the most basic example is a wire which links the devices. We are usually careful to categorise a bus by the type of information it is used to communicate. The bus which communicates addresses from the processor to the memory is called an **address bus**; the bus which communicates data values to and from the processor and memory is called the **data bus**. The buses which communicate operands and results to and from the ALU is also a **data bus**. In each case, the **bus width** is the size of information that can be communicated along the bus: a single wire can communicate 1-bit values, a 32-bit wire vector can communicate a 32-bit value.

Although there are many ways to construct a communication channel between two (or more) devices, we need to at least make the distinction between two major classes.

5.2.1 Synchronous Buses

When deploying a synchronous bus there must be a shared, unskewed clock accessible to the two devices. The devices use this clock to schedule their operations;

each operation involving signals on the bus will always take a known, integer number of **bus cycles** to complete. This fact ensures that one device can operate with a guarantee that the other will remain in step with it.

Imagine a contrived example of two devices P and Q communicating values between themselves: P sends a value x to Q, Q computes $y = x^2$ and sends y back to P, finally P checks that y is the same as $x \cdot x$ and sends 1 or 0 back to Q in the case that the test is true or false respectively. Using a synchronous bus, all these operations take say b bus cycles so we can describe our communication as follows:

Time	P	Q
$0b$	$x \rightarrow$	
$1b$		$\leftarrow y = x^2$
$2b$	$y \stackrel{?}{=} x \cdot x \rightarrow$	

Note that Q knows that the result y will be communicated at time $2b$ because the two devices are synchronised by the shared clock, so it can read this value from the bus without further control. Also note that we are using a single shared bus to do communication in both directions. The time slices allocated to each device via the clock allow this to occur without conflict.

Examples of synchronous buses within our data-path include the A, B and C buses used to communicate with the ALU. These are synchronous because their operation is governed by the system clock used to drive the processor. For example, Figure 5.2 shows the A bus being used to communicate both values read from the general-purpose register file and constant values from the control unit. These do not interfere with each other because the system clock determines the ordering of the fetch, decode and execute stages so they cannot overlap.

5.2.2 Asynchronous Buses

Devices connected by an asynchronous bus do not share a common clock and cannot therefore use it for synchronisation of their operations. Using the above example again but with an asynchronous bus, Q would not be able to make the assumption that at time $2b$ the value y would be communicated to it. That is, the value might be placed on the bus at time $2b + \varepsilon$ for some small ε in which case Q could read the wrong value. To combat this problem, extra control signals are added to the bus that allow each device to tell the other when it is ready to communicate. This introduces a **bus protocol** which must be adhered to.

As a simple example, when P wants to communicate x to Q it first raises a control signal c_1 to show it is ready to send the data. Q will have been waiting for this event to occur: as soon as it sees that $c_1 = 1$ it raises a control signal c_2 in the opposite direction to show it is read to receive the data. When P notices the change in c_2, it knows the communication can proceed. Thus, the communication of x is performed using a sequence such as:

P		Q
$c_1 = 1$	$\rightarrow$	
	$\leftarrow$	$c_2 = 1$
x	$\rightarrow$	
	$\leftarrow$	$c_2 = 0$
$c_1 = 0$	$\rightarrow$	
	$\vdots$	

Note that operations no longer need to take an integer number of bus cycles: there is no system clock, the devices are simply waiting to be triggered by events on the control signals.

An example of an asynchronous bus within our data-path is the interface between the processor and memory. Since the memory is an external device, it is unlikely that there is a shared clock between it and the processor. As a result, the devices need to enter into a protocol so that each knows when and how to operate the address and data buses that connect them. This protocol is outlined in Chapter 8.

5.3 Addressing Modes

Considering some collection of objects, **addressing** refers to identifying a particular member of the collection using a label called an **address**. As a concrete example one can imaging the 4-tuple of integers

$$A = (6,7,8,9).$$

Denoting the i-th entry of A as A_i, in order to access an element of the tuple we need to provide an index, or address, i.e., an i. For example, to access element number two we use A_2 where 2 is the address and 8 is the resulting value. An **addressing mode** encompasses the mechanism for taking an address of some sort and using it to retrieve an element from some collection. In the context of processor design, this means providing an address so that we can retrieve an operand ready to perform computation with. There are three major classes of operands: constant or immediate values, general and special-purpose register content, and memory content.

The MIPS32 ISA implies a load-store architecture. This means that instructions either access register content or memory content but not a mix of both: the `lw` and `sw` instructions are the only ones that access memory. This makes life easier since within a given instruction we limit the types of operand and hence addressing modes used. Not all ISAs are the same: traditionally Intel Pentium processor designs have included instructions that, for example, add the content of a memory location to the content of a register.

5.3.1 Immediate Addressing

An immediate operand is simply a constant value embedded into the instruction that is used in computation. For example, consider executing the addition operation $GPR[2] \leftarrow GPR[3] + 1$. The value 1 is implicit in the instruction: we do not need to fetch the value from a register or from memory, it is immediately available once the instruction is decoded. As such, calling this an addressing mode might seem somewhat overkill; we do not really provide an "address" as such since the value is available without retrieving it from somewhere.

5.3.2 Register Addressing

Addressing the general-purpose registers is a simple task: there are r registers in total so to specify which one we mean, we simply supply an address in the range $0, 1, \ldots, r-1$ which requires $\lceil \log_2(r) \rceil$ bits to represent it as an unsigned integer. This is commonly termed register direct addressing. There is a slight subtlety when it comes to accessing special-purpose registers however: we cannot simply provide an address or name, their access must be implicit in the instruction itself. As an example, the MIPS32 ISA has instructions named `mthi` and `mfhi` that move values to and from the special-purpose register HI that is used to store one half of a multiplication result. One cannot execute an instruction with the meaning $GPR[2] \leftarrow GPR[3] + HI$ since the register HI is not generally addressable. One must first execute an `mfhi` instruction to perform something like $GPR[4] \leftarrow HI$ and then execute $GPR[2] \leftarrow GPR[3] + GPR[4]$.

This represents a trade-off in the design of the ISA. If one defines the special-purpose registers as disjoint from the general-purpose registers, then more special instructions are needed to access them. If one defines them in the same set as the general-purpose registers, then our register addresses need to come from a larger range (since they need to address more registers) or the number of general-purpose registers decreases. However, the second option means that the standard instruction set can access the special-purpose registers as easily as the general-purpose registers. The trade-off is therefore roughly one of orthogonality or regularity versus the constraints of usefulness.

5.3.3 Memory Addressing

Previously, we viewed memory as an array called MEM which was accessed by providing an index or address x. The retrieval of a value at address x was denoted $MEM[x]$ while $MEM[x] \leftarrow y$ represented a store of value y into the memory location x. Here we are only interested in the address x and so consider both forms of access at once. The key thing to realise is that there are different methods, or **address-**

ing modes, by which one can construct the address x which we call the **effective address** and is passed to memory.

It is easy to start introducing more and more addressing modes to make life easier for the programmer. However, doing so is a balancing act. More addressing modes typically means a more complex processor and a requirement for the programmer or compiler to understand and use them effectively. Less addressing modes means the program must be longer since it will need to implement the behaviour of missing modes using instructions that are provided. Different processors make different trade-offs and one might encounter a huge range of different addressing modes as a result:

Direct Addressing Consider the most basic form of addressing, **direct addressing**. In this case, the effective address passed to memory is the same address as the one we want. That is, we want to access $MEM[x]$ so just feed our address x as the effective address. Very roughly, we can describe the process as a diagram:

$x = 20 \rightarrow$

Address	Value
⋮	⋮
16	
20	28
24	
⋮	⋮

$\rightarrow 28$

This is a simple addressing mode to use, and requires no addition to the standard way we interface with memory. However, the trade-off for this simplicity is that all the work in constructing the effective address x needs to be done by the programmer.

Indirect Addressing One way to extend direct addressing is to allow an indirect addressing mode instead. The memory treats an effective address as the address of the address of the element we want. You can think of this as the same way a pointer works in a C program: if the pointer is stored somewhere in memory, then to load the data it points to we need to first load the pointer and then load the data it points to. Again, we can describe this process diagrammatically as:

$x = 20 \rightarrow$

Address	Value
⋮	⋮
16	
20	28
24	
⋮	⋮

$\rightarrow$

Address	Value
⋮	⋮
24	
28	60
32	
⋮	⋮

$\rightarrow 60$

Although some processors might perform both memory accesses as the result of one instruction, and hence decrease the number of instructions the programmer needs to provide, the clear drawback to indirect addressing is the sequential nature of the two memory accesses: this can have a significant performance impact.

Indexed Addressing Perhaps a more attractive extension of direct addressing is **indexed addressing**. In this case, the effective address passed to memory is the sum of a base address and an offset. Again described diagrammatically, a memory access is performed as follows:

$$\left.\begin{array}{l}\text{base} = 16\\ \text{offset} = 4\end{array}\right\} x = 16+4 = 20 \rightarrow$$

Address	Value
⋮	⋮
16	
20	28
24	
⋮	⋮

$\rightarrow 28$

To see why this might be a useful thing to offer, consider a typical array access in a C program; for example a reference to the tenth element of an array of bytes called `A` written as `A[10]`. The element of `A` we want is ten locations on from wherever `A` is located and we know `A` is located at some fixed address. So to perform an access to `A[10]` we simply use an indexed addressing mode and set the base address to the address of `A`, say `&A`, and the offset to ten. As a result we access $MEM[\texttt{\&A} + 10]$ which is exactly what we wanted. The MIPS32 ISA uses this form of addressing mode in the `lw` and `sw` instructions: the base is allowed to be any register value while the offset is an immediate, their sum is passed to the memory as the effective address.

There is a trade-off however. We have introduced a new addressing mode that is doing some work for us by adding the base address and offset together. This is more complex than direct addressing for example, and probably requires invocation of the ALU to perform addition. On the other hand, it means programs can be written using less instructions: if we had no indexed addressing more then we would need to add the base address and offset ourselves using an extra addition instruction.

Scaled Addressing Consider what happens to the example access to byte array `A`, discussed in the context of indexed addressing, if we change the array so it holds 32-bit integers rather than bytes. The element `A[10]` is no longer ten locations on from the address of `A` but forty locations on: each element of `A` requires four locations in memory. Accessing `A[10]` using the indexed mode is now a little tricky, essentially we need to scale the offset by a factor of four to get the right address. Clearly this could be done by adding another instruction to the program, but we could also introduce a scaled addressing mode by adding a scaling factor which is applied using an inexpensive shift:

$$\left.\begin{array}{l}\text{base} = 16\\ \text{offset} = 4\\ \text{scale} = 2\end{array}\right\} x = 16 + (4 \ll 2) = 32 \rightarrow$$

Address	Value
⋮	⋮
28	
32	99
36	
⋮	⋮

$\rightarrow 99$

In the context of our example we can now still access `A[10]` using one instruction by setting the scaling factor to 2 so as to multiply the offset by 4 as above. As a result we access $MEM[\&A + 10 \cdot 4] = MEM[\&A + 40]$ as required.

5.4 Instruction Encoding

We know how to specify an opcode which tells the control unit which type of instruction to execute: this is just a number. We also know how to specify the operands of an instruction in terms of where their value can be retrieved from using some addressing mode: these are just numbers as well. Now we need to consider how these items of information are encoded, or formatted, so the control unit can understand what we mean. One can view this as an extension of the simple method from Chapter 4.

5.4.1 Instruction Selection

A major task in designing an ISA is selecting *which* instructions it will include. This will have a massive impact on many other aspects including any actual implementation. For example, one clearly needs to include hardware that matches the requirements of instructions within the ISA. There are few concrete rules for deciding which instructions should be included; for a special-purpose ISA it might suffice to exclude instructions which are extraneous for that domain but would be included in a general-purpose ISA.

However, one can quite easily start to look at the types of instructions in terms of the number of operands they use and the capabilities that result. Examining the MIPS32 ISA, it is clear that there are instructions with zero, one, two and three operands. For example, the `nop` instruction has zero operands, the `mfhi` instruction has one operand (a destination register), the `bltz` has two operands (a source register and an immediate value), the `add` instruction has three operands (two source registers and one destination register). Noting this fact seems like a triviality but in fact can have a major impact on the design of the rest of the processor and on how it is programmed.

We already have the instructions `mtlo` and `mthi` which have the meaning $LO \leftarrow GPR[x]$ and $HI \leftarrow GPR[x]$ for register operand x, and `mflo` and `mfhi` which have the meaning $GPR[x] \leftarrow LO$ and $GPR[x] \leftarrow HI$ again for some register operand x. To see the impact on having instructions with different numbers of operands, again consider having a requirement to execute the addition operation $GPR[2] \leftarrow GPR[3] + GPR[4]$. A processor that implements a 0-operand ISA is often called a **stack machine**; *everything* has to be implicit in the instruction. So to add two values together, we might invent a fictitious instruction that has the meaning $LO \leftarrow LO + HI$. Note that we do not specify the registers LO and HI in the instruction, you should think of them as being hard-coded in some sense: the addition operation cannot add values taken from any other registers. To use the instruction one would therefore need to execute the sequence

$$\begin{aligned} LO &\leftarrow GPR[3] \\ HI &\leftarrow GPR[4] \\ LO &\leftarrow LO + HI \\ GPR[2] &\leftarrow LO \end{aligned}$$

We execute four instructions in total. With one operand, things become a bit easier. Consider a processor that implements a 1-operand ISA which is often called a **accumulator machine**. To model such a processor, we introduce another fictitious instruction that has the meaning $LO \leftarrow LO + GPR[y]$ where y is the one operand we specify in the instructions, LO is again hard-coded in some sense. Now we can perform the required operation with only three instructions:

$$\begin{aligned} LO &\leftarrow GPR[3] \\ LO &\leftarrow LO + GPR[4] \\ GPR[2] &\leftarrow LO \end{aligned}$$

As one might have expected, with two operands things are even easier still. A processor that implements a 2-operand ISA is often called a **register machine**. Again we invent a fictitious instruction to model the ISA; this time the instruction means $LO \leftarrow GPR[x] + GPR[y]$ where x and y are our two specified operands. The required operation can now be executed in two instructions:

$$\begin{aligned} LO &\leftarrow GPR[3] + GPR[4] \\ GPR[2] &\leftarrow LO \end{aligned}$$

Finally, with a 3-operand ISA (implementations of which are usually still classed as register machines) we can perform the required operation with just one instruction:

$$GPR[2] \leftarrow GPR[3] + GPR[4]$$

Notice the trade-off which occurs. The more operands in our instructions the less instructions are typically needed to execute an operation; however, we need to specify all these operands so our instructions may be bigger. The less operands we have,

```
struct Rtype
{
  uint32 funct  :  6, // ALU function
         shamt  :  5, // ALU shift distance
         Rd     :  5, // register address 2
         Rt     :  5; // register address 1
         Rs     :  5; // register address 0
         opcode :  6; // opcode
}

struct Itype
{
  uint32 imm    : 16, // immediate
         Rt     :  5, // register address 1
         Rs     :  5, // register address 0
         opcode :  6; // opcode
}

struct Jtype
{
  uint32 off    : 26, // offset
         opcode :  6; // opcode
}
```

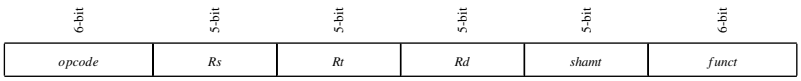
Listing 5.1 The three standard MIPS32 instruction formats (as C structures).

the more information is provided implicitly in the instruction rather than explicitly as operands. Thus the instructions can be smaller.

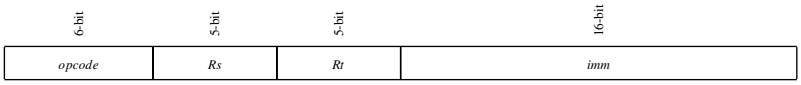
Just as the number of operands has an impact on things, so does the type of those operands. In the above example we are only using register operands whereas in the actual ISA we have immediate operands and register operands. By collating the different instruction types, we can classify the instructions in the reduced MIPS32 ISA as follows:

3 instruction with 0 register operands + 0 immediate operands
5 instructions with 1 register operand + 0 immediate operands
1 instruction with 2 register operands + 0 immediate operands
9 instructions with 3 register operands + 0 immediate operands
1 instruction with 0 register operands + 1 immediate operand of 16 bits
2 instruction with 0 register operands + 1 immediate operand of 26 bits
4 instructions with 1 register operand + 1 immediate operand of 16 bits
5 instructions with 2 register operands + 1 immediate operand of 16 bits

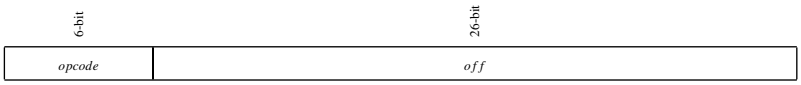
Clearly we can reduce these seven **instruction classes** to even less; if we have a class of three register operand instructions, this can also accommodate those with two, one and zero register operands simply by ignoring those operands which are not needed.



(a) An R-type or "register" instruction format.



(b) An I-type or "immediate" instruction format.



(c) A J-type or "jump" instruction format.

Figure 5.3 The three standard MIPS32 instruction formats (as bit-fields).

5.4.2 Instruction Formats

An **instruction format** is basically a template which can be used to encode a class of instructions. Within the template the concrete opcode and operands are placed into fields to form an encoded instruction. The MIPS32 ISA uses three different instruction formats, called the R-type, I-type and J-type formats, and which are detailed in Figure 5.3 and Listing 5.1. These formats correspond to the actual encodings in Table 5.1 in the sense that it shows which fields of a format should be filled with which information to encode a given instruction.

5.4.3 Basic Encoding and Decoding

Consider encoding an `add` instruction to represent our running example which performs $GPR[2] \leftarrow GPR[3] + GPR[4]$. The `add` instruction uses an R-type format; to construct the instruction we first fill the `opcode` and `funct` fields with the 6-bit constants $000000_{(2)}$ and $100000_{(2)}$ respectively. These fields combine to tell the control unit that this is an `add` instruction. Then, we fill the register operand fields `rs`, `rt` and `rd` with the constants $00011_{(2)}$, $00100_{(2)}$ and $00010_{(2)}$ respectively and zero the remaining blank field. The end result is the 32-bit number

```
uint32 encode_Rtype( int opcode, int rs, int rt, int rd, int funct )
{
  return ( ( opcode & 0x3F ) << 26 ) |
         ( ( rs     & 0x1F ) << 21 ) |
         ( ( rt     & 0x1F ) << 16 ) |
         ( ( rd     & 0x1F ) << 11 ) |
         ( ( funct  & 0x3F ) <<  0 ) ;
}
```

Listing 5.2 A simple C function to encode a MIPS32 R-type instruction.

```
task decode_Rtype;
  input  [31:0] instruction;

  output [ 5:0]      opcode;
  output [ 4:0]          rs;
  output [ 4:0]          rt;
  output [ 4:0]          rd;
  output [ 5:0]       funct;

  begin
    opcode = instruction[31:26];
    rs     = instruction[25:21];
    rt     = instruction[20:16];
    rd     = instruction[15:11];
    funct  = instruction[ 5: 0];
  end
endtask
```

Listing 5.3 A simple Verilog task to decode a MIPS32 R-type instruction.

$$00000000011001000001000000100000_{(2)}$$

that represents our instruction. Filling in these fields is somewhat tricky so one would normally utilise some form of function to make things easier; Listing 5.2 details a simple C function which will return an encoded R-type instruction given a list of opcode and operand values. It first masks each value so that it is the right size and then shifts them into the right place; the parts are ORed together to form the end result.

Decoding an instruction is more or less the exact reverse process to encoding it: we simply extract information from the right place. However, the three MIPS32 instruction formats make the decoding process much easier than it might otherwise be. In particular, they provide a number of advantages typical of many instruction encoding strategies:

1. All instructions are of a fixed length; this means we never need to perform more than one memory access to retrieve them. The control unit can be simple since, as a result, it does not need to decide when and if more parts of an instruction are required.
2. Common fields between the formats are aligned in the same place, for example the opcode is always the most-significant six bits. This allows for more efficient decoding since the control unit never needs to conditionally shift around the fields

```
1 function ext;
2   input [31:0] instruction;
3
4   begin
5     extend_16bit = instruction[15] ? { 16'hFFFF, instruction[15:0] } :
6                                      { 16'h0000, instruction[15:0] } ;
7   end
8 endfunction
```

Listing 5.4 A simple Verilog task to decode a 16-bit sign-extended immediate value.

to get the right information: it just extracts the bits required from a known location.

As such, it is easy to write Verilog tasks that could be used within the control unit to decode instructions. Listing 5.3 details an example which could decode our previously encoded R-type `add` instruction. Since Verilog offers a convenient mechanism to describe the extraction of bits, the task amounts only to a few assignments.

However, things are somewhat complicated when we consider immediate values in the I-type and J-type formats: remember such immediates might represent positive or negative numbers. Consider, for example, how to decode the 16-bit immediate value in the `addi` instruction; naively one might simply extract the bottom 16 bits of the instruction. The problem comes when this value is used as a 32-bit operand. The value $-1_{(10)}$ would be encoded as the 16-bit twos-complement constant

$$FFFF_{(16)}.$$

When naively padded with zeros to form a 32-bit value ready for performing computation, we get

$$0000FFFF_{(16)}$$

which is no longer $-1_{(10)}$ but $65535_{(10)}$. The problem is that we need to pad with the most-significant bit, or sign bit since this dictates the sign of the number, not zeros. Application of this process is called **sign extension** and produces the constant

$$FFFFFFFF_{(16)}$$

which is still $-1_{(10)}$ as a 32-bit value. The sign extension process is also easy to describe in Verilog; Listing 5.4 gives an example. The MIPS32 ISA description in Table 5.1 denotes the application of the sign extension process to an immediate operand x as $ext(x)$. It is important to realise that not all immediate values need be sign extended. For example, shift distances are *always* positive so may not require sign extension but rather just naive extraction.

000000	*Rs*	*Rt*	*Rd*	00000	100000
111111 *opcode*	*Rs* *Rs*	*Rt* *Rt*	*Rd* *Rd*	00000 *shamt*	100000 *funct*
000000	00000	00000	00000	00000	100000

Figure 5.4 A fictitious example of variable length encoded instructions: a 1-word `add` instruction (with three register operands) is followed by a 2-word instruction (with six register operands), and finally by a 1-word `nop` instruction (with no operands).

5.4.4 More Complicated Encoding Issues

The MIPS32 encoding mechanism is typical of 32-bit processors in that the instruction size is large enough to comfortably accommodate all the required operands in a few "friendly" formats. Different designs require different approaches however; an instruction encoding mechanism for 8-bit instructions cannot use the same approach for example, since space is so limited. Although there are a huge range of options, we present a few techniques below that one might use. All of the presented techniques imply a trade-off in terms of their use: we might save space or pack more information into the same space but will inevitably complicate the control unit as a result. This is because the decoding step will become harder and less regular since special cases will need to be checked for and accommodated.

5.4.4.1 Fixed and Variable Length Encodings

Roughly speaking, **code density** relates to the amount of memory taken up by the instructions for a program; higher code density, i.e., number of instructions or amount of computation carried out per unit space, means the program uses less space. We have already encountered a design decision that can impact on code density when we discussed different instruction types: the more operands an instruction has, the larger it is but typically the less instructions are actually needed to perform an operation. In an attempt to get the best of both worlds from this trade-off, some designs use what is called a **variable length encoding**. Previously, we looked at the MIPS32 encoding mechanism where every instruction was the same length. A variable length encoding breaks this constraint by allowing instructions that require many operands to be longer than those that require few operands. As a result, one can achieve high code density while still having many operands.

One way to achieve this is to reserve one value of the `opcode` field as some form of "escape code" which specifies the need to fetch more instruction from memory. Figure 5.4 attempts to illustrate this in the context of the MIPS32 instruction formats already presented. There are three instructions arranged in memory starting at some address. The first is a standard 1-word `add` instruction which can specify three

register addresses as operands. The second is some fictitious 2-word instruction; the first part uses the value $111111_{(2)}$ as the opcode to show that we need to fetch another word from memory to complete the instruction. The second part specifies the actual opcode with both parts containing three register address operands (a total of six operands). Finally, the third instruction is a standard 1-word `nop`.

In this contrived example one can clearly see the trade-off offered by having a variable length encoding. That is, we can now accommodate instructions with many operands, and still maintain good code density, but doing so means we might need two accesses to memory to load such instructions. Our control unit will naturally be more complicated as a result.

5.4.4.2 Compact Instruction Encodings

In the above discussion we used the concept of an escape code to denote that one opcode meant something special while all others could be interpreted as normal. We can re-use this concept to look at the issue of how one might design an instruction encoding mechanism for very compact instructions. That is, instructions that are perhaps only 8-bits long.

Consider a fictitious ISA with 8-bit instructions; it has four registers and a range of instructions that all fall into one of the following classes:

0 register operand + 0 immediate operands
1 register operands + 0 immediate operands
2 register operands + 0 immediate operands
0 register operands + 1 immediate operand of 7 bits
1 register operand + 1 immediate operand of 4 bits

Given register addresses can be represented as as 2-bit values, a possible encoding might define five instruction formats:

1. $0xxxxxxx_{(2)}$
2. $10rrxxxx_{(2)}$
3. $11ccrrrr_{(2)} \rightarrow 1100rrrr_{(2)} \ldots 1110rrrr_{(2)}$
4. $1111ccrr_{(2)} \rightarrow 111100rr_{(2)} \ldots 111110rr_{(2)}$
5. $111111cc_{(2)} \rightarrow 11111100_{(2)} \ldots 11111111_{(2)}$

where c represents a field for opcodes, x represents space for an immediate and r represents space for a register address. From this we can see there are three possible two-operand instructions, three possible 1-operand instructions and four possible 0-operand instructions. Notice that in each case we are using an escape code to mark a different format. For example, $1100rrrr_{(2)}$, $1101rrrr_{(2)}$ and $1110rrrr_{(2)}$ are all two-operand instructions in the third class. However, we use what would have been the instruction $1111rrrr_{(2)}$ to signal that actually we mean something else: that is, one needs to read more opcode and hence translate into the fourth class of 1-operand instruction of the form $1111ccrr_{(2)}$.

One might view this as an application of variable length opcodes, rather than the variable length instructions as described previously; it is commonly called the **expanding opcode** approach. The key thing to realise is that even when we are allowing parts of the opcode to occur in various different places, every instruction is uniquely identifiable. This was trivial to see when looking at the MIPS32 ISA since the opcode was a fixed field. Here it is harder to see but this fact is vital if the encoding is to work correctly: if instructions are not uniquely identifiable, the control unit has no hope of telling which one we meant !

5.4.4.3 Compressing Register Addresses

Clearly space is at a premium as far as instruction encodings go, and in order to maximise the utilisation of this space, an ISA generally provides a general purpose register file whose size is a power-of-two. This means with n-bit register addresses we can address all 2^n registers. This has advantages and disadvantages. Although it ensures we get the best use of space since no bits are wasted, if we want less than 2^n registers, the next size down is 2^{n-1}. If we start with thirty two registers, the next size down is sixteen: quite a jump. In ISAs that are designed for implementation in a context where resources are constrained, using less registers makes a lot of sense. But this must be balanced against performance so what happens if we want an ISA with twelve general-purpose registers for example, can we do better than use 4-bit addresses which could address sixteen registers and is hence wasteful ?

If we assume there are formats for 3-operand instructions, using 4-bit register addresses would total twelve bits of operand in such an instruction; let us denote these three 4-bit register addresses as $A_{3...0}$, $B_{3...0}$ and $C_{3...0}$. The crucial thing to realise is that because

$$A \in \{0_{(10)}, 1_{(10)} \dots 11_{(10)}\}$$

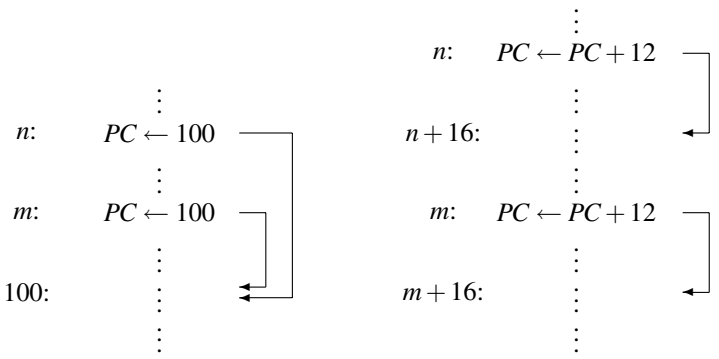
then the top two bits of A, i.e., $A_{3...2}$, cannot be $11_{(2)}$. That is, we always have that

$$A_{3...2} \in \{00_{(2)}, 01_{(2)}, 10_{(2)}\}.$$

So we can store the bottom two bits $A_{1...0}$ as normal and then use the fact that there is an unused state in the top two bits to save space. Since we have A, B and C to encode we have that

$$\begin{aligned} A_{3...2} &\in \{00_{(2)}, 01_{(2)}, 10_{(2)}\} \\ B_{3...2} &\in \{00_{(2)}, 01_{(2)}, 10_{(2)}\} \\ C_{3...2} &\in \{00_{(2)}, 01_{(2)}, 10_{(2)}\}. \end{aligned}$$

There are thus only twenty seven possible combinations which can be encoded in only five bits, since $2^5 > 27$, rather than the six which we would have used in the naive case. In total then, we need six bits for the bottom halves of the three register addresses and five bits to encode the top halves; this saves us one bit which can be used elsewhere, e.g., to double the number of opcodes we can specify.



(a) Absolute branches. (b) Offset branches.

Figure 5.5 The difference between absolute and offset (or *PC* relative) branches.

5.5 Control-Flow

So far we have only considered programs, i.e., sequences of instructions, whose **control-flow** is a straight line. That is, the flow of execution starts at the top and proceeds through each instruction in sequence before ending at the bottom. In reality, this places a massive restriction on the types of programs we can write. For example we cannot conditionally execute sequences of instructions based on some input or intermediate result, or iterate over a sequence of instructions to form what we commonly call **loops**. Providing the ability to alter the control-flow of a program is therefore a vital part of both instruction set and micro-architecture design. This is particularly true since one can view instructions that alter control-flow as pure overhead: they are not really performing any useful computation themselves, just selecting which computation is performed. As such, ensuring that the control-flow mechanism does not have a significant impact on performance is also vital.

Roughly speaking, an instruction which might alter the flow of execution is called a **branch** instruction. Any branch will achieve a change in control-flow by updating the value of *PC*, the program counter, which holds the address of the next instruction to be executed. We can classify branches according to a number of criteria which dictate how and when they perform this update:

Conditional Versus Unconditional A branch is unconditional if it *always* updates the program counter; such a branch is sometimes called a **jump**. A branch is conditional if the update only occurs when some condition is true. Typically the condition will be a comparison between operands. When a branch is executed so that we know where next to fetch instructions from, i.e., the processor knows if it is taken or not, we say the branch has been **resolved**.

Examples of both forms are easy to see in the MIPS32 ISA. For example, the `j` instruction is an unconditional branch, it simply sets the *PC* to equal the immediate operand. Both `beq` and `bltz` are conditional branches since they compare two values to each other, two registers or a register and an immediate respectively, and only update the *PC* if the result of the comparison is true.

Absolute Versus Relative An absolute branch sets the *PC* to a value unrelated to the value it currently holds; the instruction simply replaces the value in *PC* with some new address. A *PC* relative (or offset) branch updates the *PC* by changing it by some amount (i.e., adding or subtracting some value),; the end result is that the branch is relative to the *PC* rather than just any address.

Most branches are fairly local, it is uncommon for a branch to update the *PC* with an absolute address. As such, one can encode the target address more compactly as an offset from the *PC* than a full 32-bit address. This means that, for example, a single MIPS32 conditional offset branch `beq` can include a 16-bit immediate and cope with most offsets required.

There is one caveat to offset branches, which is related to *when* the branch is actually executed. In the fetch-decode-execute cycle we already said that the *PC* would be incremented by some amount during the fetch stage so it points to the next instruction to execute. By the time the processor enters the execution phase for a taken branch, we are essentially saying that this is not actually the instruction we want to execute at all, some other one is. But the *PC* has already been incremented so we need to take this into account when calculating the offset: typically the offset required will be one instruction, four bytes in the MIPS32 case, less than one might have thought. If this off-by-one problem is not taken into account, the processor will branch to the wrong place !

Figure 5.5 attempts to demonstrate the difference between absolute and offset branches; both Figure 5.5a and Figure 5.5b have two identical branches at addresses *n* and *m*, the result of their execution is different however. In particular, both branches in Figure 5.5a result in control-flow being sent to address 100 while in Figure 5.5b the branch target depends on what address the branch instruction resides at.

Immediate Versus Computed The memory in a typical 32-bit processor has 2^{32} addressable locations. To branch to an arbitrary address therefore, one has to provide a 32-bit address. Encoding a 32-bit address inside a 32-bit instruction is impossible: there is no space for the opcode so some compromise must be made. The unconditional absolute branch `j` is a good example of this, it sets the *PC* to equal a 26-bit immediate which cannot address the whole memory but does cover a fairly large region.

To cope with the need to branch to arbitrary addresses, most ISAs include a computed branch instruction which takes the new *PC* value from a register operand. Since the register can contain a full 32-bit value, the branch can be to anywhere in memory. The trade-off is that one has to execute more instructions in order to compute this address in the register before the branch is executed; with immediate branches these extra instructions are not required.

Other than these different patterns of actual behaviour, the rest of the fetch-decode-execute cycle is unchanged. The completion of a branch in the execution stage means the right value is in the *PC* for the next fetch stage; the cycle simply carries on as normal except that we have altered where in the program we are now fetching instructions from.

One slight subtlety introduced by the MIPS32 ISA is the idea that we need not store the whole branch target as an immediate in the instruction encoding. Each instruction is a 32-bit value so if the *PC* starts at address zero, instructions always start on a multiple of four bytes; they are word aligned. If we are using a byte addressable memory where each byte has an address, the addresses of instructions are all 0 modulo 4, which is to say the least-significant two bits of the address are always zero. Since the *PC* is always word aligned, any offset from it will also be a multiple of four bytes and hence will also always have the least-significant two bits as zero. Given this fact, we do not need to store the bits in an encoded instruction. For example, the meaning of the `j` instruction is

$$PC \leftarrow (imm : 00_{(2)})$$

where *imm* is a 26-bit immediate stored in the instruction. This 26-bit immediate has the two implicit zero bits appended to the least-significant end via the concatenation operator before it is used to update the *PC*. This way, our 26-bit immediate does not waste space, and can specify a larger range of addresses as a result.

5.5.1 Predicated Execution

In the MIPS32 ISA, redirection of control-flow is done explicitly using dedicated branch instructions. Use of **predicated** or **guarded** execution offers an alternative to this model by associating a predicate function to each instruction and only executing the instruction if the predicate is satisfied, i.e., evaluates to true. Rather than set the *PC* to some new value, this approach allows one to "skip over" instructions based on the associated predicate. It is ideal for conditional execution of short sections of code due to the low overhead associated. Since MIPS32 does not support predicated execution, to show how the concept works we need several extensions: these are imaginary, but relate closely to the form of instruction set such as ARM.

Imagine we extend MIPS32 to include a carry flag *CF* which resides in the processor status register: this flag is set by arithmetic instructions such as addition when a carry-out is produced. If we also include a branch instruction based on the value of *CF*, we can write a short fragment

$$\begin{aligned} & \ldots \\ & GPR[2] \leftarrow GPR[3] + GPR[4] \\ & \text{if } CF = 0, PC \leftarrow skip \\ & GPR[5] \leftarrow GPR[5] + 1 \\ skip : & \ldots \end{aligned}$$

that performs an addition, then tests if the carry flag CF was set in the status register: if the sum of $GPR[3]$ and $GPR[4]$ is too large to fit into 32 bits, a carry-out is produced and CF will be set. If the carry flag is set, a second addition instruction adds the carry to $GPR[2]$; if the carry flag is not set, the branch instruction skips over this second addition. The fragment results in two or three instructions being executed depending on if the branch is taken or not.

Now imagine we extend MIPS32 further by allowing instructions to specify a predicate: *every* instruction can either be executed or skipped based on the value of CF. One might re-write the fragment from above as follows:

$$\begin{aligned} &\ldots \\ &GPR[2] \leftarrow GPR[3] + GPR[4] \\ &\text{if } CF = 1, GPR[5] \leftarrow GPR[5] + CF \\ skip: &\ldots \end{aligned}$$

Now the second instruction has a predicate associated with it, only when the carry flag is set does this instruction get executed; if the predicate is not satisfied, then the processor just skips over it. This sequence is now more compact since it is only two instructions long: roughly speaking the processor executes only one or two instructions depending on if the predicate on the second instruction is satisfied. The trade-offs in deploying a system for predicated execution are quite involved. Typically programs using predicated execution exhibit better code density and, since they contain less branches, thus get more concrete computation done in the same time. On the other hand the predicates take up precious space in the instruction encoding and the control unit needs to be slightly more complicated to cope with skipping over instructions rather than passing them through the execute stage of the fetch-decode-execute cycle.

To better demonstrate the advantages of predicated execution, consider a more realistic example; the canonical such example is an implementation of the **Greatest Common Divisor (GCD)** algorithm due to Euclid. The idea is that given two integers a and b, the algorithm finds the largest integer c which exactly divides both a and b; the case where $c = 1$ means a and b are co-prime. Figure 5.6a shows the Euclidean algorithm written in C. To implement this function we will again extend MIPS32 to include a rich set of flags in the status register which are set by a special compare instruction. Specifically, imagine the execution of

$$GPR[x] \stackrel{?}{=} GPR[y]$$

results in

- the EQ flag being set when $GPR[x] = GPR[y]$,
- the NE flag being set when $GPR[x] \neq GPR[y]$,
- the LT flag being set when $GPR[x] < GPR[y]$,
- the LE flag being set when $GPR[x] \leq GPR[y]$,
- the GT flag being set when $GPR[x] > GPR[y]$,
- the GE flag being set when $GPR[x] \geq GPR[y]$.

```
int GCD( int a, int b )
{
  while( a != b )
  {
    if( a > b )
      a -= b;
    else
      b -= a;
  }

  return a;
}
```

(a) The GCD function written in C.

$$
\begin{array}{ll}
 & \ldots \\
loop: & GPR[2] \stackrel{?}{=} GPR[3] \\
 & \text{if } EQ = 1, PC \leftarrow exit \\
 & \text{if } LE = 1, PC \leftarrow skip \\
take: & GPR[2] \leftarrow GPR[2] - GPR[3] \\
 & PC \leftarrow loop \\
skip: & GPR[3] \leftarrow GPR[3] - GPR[2] \\
 & PC \leftarrow loop \\
exit: & \ldots
\end{array}
$$

(b) The GCD function implemented using branches.

$$
\begin{array}{ll}
 & \ldots \\
loop: & GPR[2] \stackrel{?}{=} GPR[3] \\
 & \text{if } GT = 1, GPR[2] \leftarrow GPR[2] - GPR[3] \\
 & \text{if } LE = 1, GPR[3] \leftarrow GPR[3] - GPR[2] \\
 & \text{if } NE = 1, PC \leftarrow loop \\
exit: & \ldots
\end{array}
$$

(c) The GCD function implemented using predicated instructions.

Figure 5.6 A C function and ARM assembly language implementation of the Euclidean algorithm for computing GCD.

and that the flags are *only* set by the comparison instruction (i.e., not overwritten by execution of arithmetic). Again these are imaginary extensions, but again they closely follow real instruction sets such as ARM.

Figure 5.6b demonstrates a standard approach to implementing the function body under the assumption that `a` and `b` are stored in the registers $GPR[2]$ and $GPR[3]$. The approach is controlled by comparing $GPR[2]$ and $GPR[3]$, i.e., `a` and `b`, and then using branch instructions to direct control-flow to the correct case: if $GPR[2] = GPR[3]$, then the first branch exits the loop, if $GPR[2] \leq GPR[3]$ then the second

branch directs control-flow to the instruction which subtracts `a` from `b` and then branches back to the start of the loop. Otherwise control-flow drops through to the instruction which subtracts `b` from `a` and then branches back to the start of the loop. In contrast, Figure 5.6c eliminates those branches by reimplementing the function body using three predicated instructions. Notice that the only branch is that which transfers control to the start of the loop: the branches required to select between the assignments `a = a - b` or `b = b - a` are managed by predicated execution. For example, the first subtract instruction is predicated on the *GE* or greater than or equal flag: if the comparison between `a` and `b` meant that `a` was greater than or equal to `b`, the *GE* flag would be set and hence the subtraction instruction (i.e the assignment `a = a - b`) would be executed, otherwise it is skipped.

5.5.2 Function Calls

Function calls are clearly another type of control-flow mechanism so we briefly look at the issue here with respect to support that is provided by the ISA. Considering a call to a sub-routine X, i.e., a function X with no arguments, execution looks like the following:

1. Branch to the sub-routine X, i.e., call the sub-routine.
2. Execute the sub-routine X.
3. Branch back to just after the call instruction.

The first two steps are simple enough and can be catered for using instructions already presented. The problem comes in the third step: a given sub-routine may be called from a number of different places, how do we know where to branch back to ? The answer is to store the address to branch back to, the **return address**, somewhere when we call the sub-routine. There are three main options as regards where to store the return address:

- One could store the return address in a single fixed memory location or register. Although simple, this is a bad choice as it only allows the call of one sub-routine at a time; nested sub-routines are not really possible.
- One could store the return address in a per-sub-routine fixed memory location or register. This is a better option in that it allows nested sub-routine calls but still does not allow any form of recursion.
- Finally, one could use a stack; this is the most common solution since it allows total freedom with recursion and also facilitates argument passing when considering functions rather than simply sub-routines.

The MIPS32 ISA includes a common instruction called `jal` for jump-and-link that helps implement the first option above. The `jal` instruction performs two tasks: firstly it stores the return address in $GPR[31]$, and then it performs a normal, unconditional absolute branch. The `jr` instruction offers a mechanism to branch to the contents of a register so this enables the sub-routine to branch back to the return

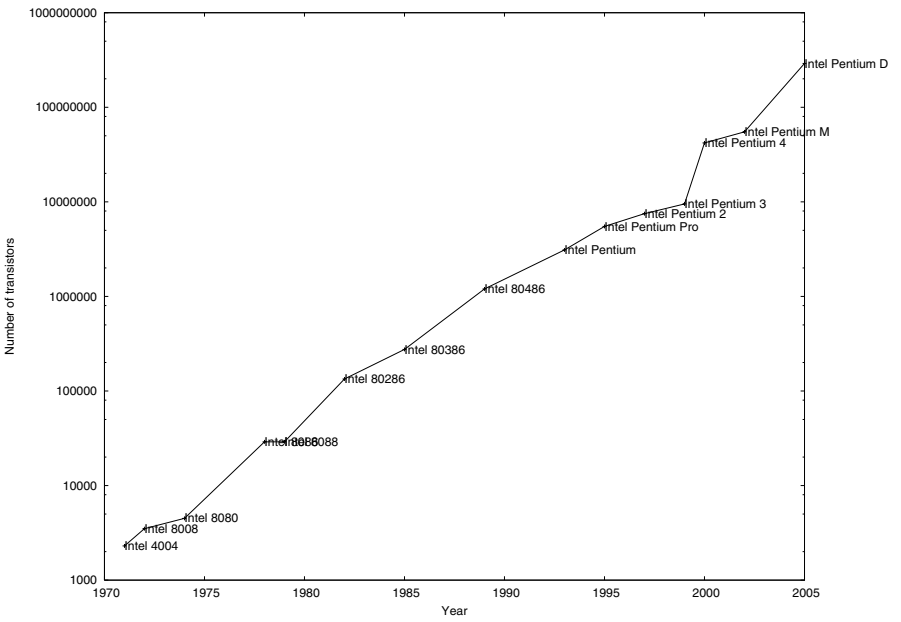


Figure 5.7 A timeline of Intel processor innovation demonstrating Moore's Law (data taken from `http://www.intel.com/technology/mooreslaw/`).

address by using *GPR*[31]. As such, these two instructions offer an efficient but limited form of sub-routine call. We will examine a more complete mechanism which solves the problems in Chapter 11 since more general function calls are essentially a mechanism implemented by the compiler rather than in the processor. Note that the `jal` instruction is the first we have seen to blur the divide between general and special-purpose registers: *GPR*[31] is a general-purpose register but it is reserved here for a special role.

5.6 Some Design Philosophy

5.6.1 Moore's Law

Gordon Moore, co-founder of Intel, is credited with identifying an important trend with respect to development of transistor-based technology: roughly speaking, Moore's Law [46] says that the number of transistors that one can fit in unit area roughly doubles every two years. This has become a form of a self-fulfilling prophecy however in the sense that that the "law" is now an accepted truth so indus-

try is forced to deliver improvements: industry is driven by the law rather than the other way around ! Figure 5.7 demonstrates the manifestation of Moore's Law on the development of Intel processors.

Of course Moore's Law is not a "law" at all but has held for the last fourty or so years all the same. The implications for design of a processor can be viewed in (at least) two ways:

- If one can fit more transistors in unit area, the transistors are getting smaller and hence working faster due to their physical characteristics. As a result, one can take a fixed processor design and over time it will get faster or use less power as a result of Moore's Law.
- If one can fit more transistors in unit area, then one can design and implement more complex structures in the same fixed area. As a result, over time one can use the extra transistors to improve the processor design yet keep it roughly the same size.

The second option above is clearly more reasonable in the long term. That is, since there are physical limits that dictate how small and fast a transistor can be, better design is always the best long-term option. However, even in this case Moore's Law can have an impact on thinking. For example, since the development of a new processor takes many years, Moore's Law can invalidate a given design quite quickly: what is the best design one day might no longer be the best with respect to a new generation of improved technology two years later. Furthermore, not all components obey Moore's Law. In particular, the rate at which memory access speed has improved does not match the rate at which memory density has improved; this leads to a widening disparity between processor and memory speed which can be a significant problem.

5.6.2 RISC versus CISC

Many of the trade-offs presented so far have been a matter of the length and complexity of programs, and hence performance in a rough sense, versus simplicity in design and implementation. This is a classic trade-off and one which has resulted in two schools of design methodology, the so-called **Reduced Instruction Set Computers (RISC)** and **Complex Instruction Set Computers (CISC)**.

RISC Put simply, RISC designs attempt to provide an ISA where instructions all do very simple operations. To write a program using such an ISA might take more instructions since they are so simple, but this same simplicity means a processor implementation can execute them very quickly; each instruction typically takes one clock cycle. RISC designs are characterised by the provision of many general-purpose registers, few addressing modes and a fixed length, regular instruction encoding. This makes it easy to program for both humans and compilers; performance of compiler-generated code can often match or better that of human-generated code.

CISC In contrast, CISC designs offer instructions which perform much more involved operations. For example, a CISC ISA might include an instruction for computing square roots while a RISC design would demand you constructed an algorithm in software using only the basic instructions available. Although the CISC instruction set will be harder to implement and instructions might take longer to execute, a program will typically be shorter since there is more work being done by each instruction. Instructions might take many clock cycles to complete. A CISC design will typically have fewer general-purpose registers (perhaps using some of them for special-purposes), many addressing modes and might use a variable length instruction encoding. As a result, it will be harder for a compiler to generate efficient code.

Like many areas of computer science, research into ways of improving CISC designs such as the Intel 80486 and Motorola 68000 was initially undertaken by both industry and academia. Early experiments by IBM [17] into the removal of many under-utilised and over-designed features eventually led to the POWER and PowerPC RISC architectures. At the same time, research in Berkeley [50] and Stanford [27] heavily influenced the MIPS and SPARC RISC architectures. These processors command a significant market share; in particular they dominate the games console and embedded arenas.

Ultimately there is no right or wrong answer to the question of which is the better approach, it depends entirely on the design context and goals. However, since initial research and subsequent commercialisation in the late 1970s and early 1980s, RISC design has dominated the field of computer architecture. Even processors which were conventionally CISC-based, such as the Intel Pentium range, have gradually migrated to using an internal RISC core behind a translation layer for backward compatibility. Like most aspects of computer architecture, successful RISC designs rely heavily on selecting an effective trade-off between many competing factors. Such decisions are typically resolved by performing a workload characterisation of the sorts of program that the processor will execute. By considering an average program, the architecture is designed so that the average case is optimised while non-typical cases are marginalised or omitted.

5.7 Putting It All Together

As described, developing a Verilog processor model that implements the reduced MIPS32 ISA might seem a daunting task. Although we defer a detailed investigation of the ALU and main memory to Chapter 7 and Chapter 8, the goal of this section is to show that for the most part, such a processor model is simply a big state machine. As such, the resulting model could be more accurately termed a simulator or emulator: it does the right thing functionally but there is not enough detail to generate realistic hardware from the model. We make a further simplification by ignoring system calls, interrupts and the like until we encounter and explain them in Chapter 12.

```
1  `define STATE_FETCH     3'b000
2  `define STATE_DECODE    3'b001
3  `define STATE_EXECUTE   3'b010
4  `define STATE_MEMORY    3'b011
5  `define STATE_WRITE     3'b100
6
7  `define OPCODE_SPECIAL1 6'b000000
8  `define OPCODE_SPECIAL2 6'b011100
9  `define OPCODE_ADDI     6'b001000
10 `define OPCODE_LW       6'b100011
11 `define OPCODE_SW       6'b101011
12 `define OPCODE_BEQ      6'b000100
13 `define OPCODE_BNE      6'b000101
14 `define OPCODE_BLEZ     6'b000110
15 `define OPCODE_BGTZ     6'b000111
16 `define OPCODE_REGIMM   6'b000001
17 `define OPCODE_J        6'b000010
18 `define OPCODE_JAL      6'b000011
19
20 `define FUNCT_SYSCALL   6'b001100
21 `define FUNCT_BREAK     6'b001101
22 `define FUNCT_ADD       6'b100000
23 `define FUNCT_SUB       6'b100010
24 `define FUNCT_MUL       6'b000010
25 `define FUNCT_MULT      6'b011000
26 `define FUNCT_AND       6'b100100
27 `define FUNCT_OR        6'b100101
28 `define FUNCT_XOR       6'b100110
29 `define FUNCT_NOR       6'b100111
30 `define FUNCT_SLLV      6'b000100
31 `define FUNCT_SRLV      6'b000110
32 `define FUNCT_MFHI      6'b010000
33 `define FUNCT_MFLO      6'b010010
34 `define FUNCT_MTHI      6'b010001
35 `define FUNCT_MTLO      6'b010011
36 `define FUNCT_JR        6'b001000
37
38 `define RT_BLTZ         5'b00000
39 `define RT_BGEZ         5'b00001
40
41 module processor( input  wire clk,
42                   input  wire rst );
43
44   reg  [31:0] PC;
45   reg  [31:0] IR;
46   reg  [31:0] HI;
47   reg  [31:0] LO;
48
49   reg  [ 7:0] MEM[0:1023];
50   reg  [31:0] GPR[0:  31];
51
52   reg  [ 2:0] state;
53
54   reg  [ 5:0] dec_opcode;
55   reg  [ 4:0] dec_rs;
56   reg  [ 4:0] dec_rt;
57   reg  [ 4:0] dec_rd;
58   reg  [ 5:0] dec_funct;
59
60   reg  [31:0] dec_imm16;
61   reg  [31:0] dec_imm26;
62   reg  [31:0] dec_imm16s;
63   reg  [31:0] dec_imm26s;
64   reg  [31:0] dec_ext16;
65   reg  [31:0] dec_ext26;
66   reg  [31:0] dec_ext16s;
67   reg  [31:0] dec_ext26s;
68
69   reg  [31:0] alu_dst_lo;
70   reg  [31:0] alu_dst_hi;
71
72   ...
73
74 endmodule
```

Listing 5.5 The interface to, and definitions for and within, a Verilog module that models a MIPS32-based processor.

```
1   always @( posedge rst )
2   begin
3     PC = 0;
4     IR = 0;
5     HI = 0;
6     LO = 0;
7
8     state = `STATE_FETCH;
9
10    $readmemh( "processor.bin", MEM, 0 );
11  end
12
13  always @( posedge clk )
14  begin
15    if( !rst )
16    begin
17      case( state )
18        `STATE_FETCH   : begin
19                           fetch;
20                           state = `STATE_DECODE;
21                         end
22        `STATE_DECODE  : begin
23                           decode;
24                           state = `STATE_EXECUTE;
25                         end
26        `STATE_EXECUTE : begin
27                           execute;
28                           state = `STATE_MEMORY;
29                         end
30        `STATE_MEMORY  : begin
31                           memory;
32                           state = `STATE_WRITE;
33                         end
34        `STATE_WRITE   : begin
35                           write;
36                           state = `STATE_FETCH;
37                         end
38      endcase
39    end
40  end
```

Listing 5.6 Verilog processes that determine the processor behaviour.

Listing 5.5 describes the simple interface to the processor; it just includes external clock and reset signals. Normally this interface would include a lot of other connections with external devices, the main memory being the most obvious. However, we have not covered that material yet so we model the memory as an internal 2-dimensional Verilog register to the processor for now. The processor is driven by the processes in Listing 5.6. The first process resets the processor whenever a positive edge on the reset signal `rst` is detected. The reset procedure involves zeroing the registers *PC*, *IR*, *HI* and *LO*, initialising the state machine,represented by `state`, and then initialising the memory using the `$readmemh` system task: you can think of this as "magic" which would be replaced by an operating system running on the processor tasked with loading and starting execution of programs. The second task acts to update the state machine each time a positive edge on the clock signal `clk` is detected; the faster the clock works, the faster the processor works. The state machine simply iterates around the fetch-decode-execute cycle calling the right task at

```
task alu( input [ 5:0] alu_oper,
          input [31:0] alu_src0,
          input [31:0] alu_src1 );
begin
  case( alu_oper )
    `FUNCT_SYSCALL : ;
    `FUNCT_BREAK   : ;
    `FUNCT_ADD     :  alu_dst_lo  =  (alu_src0 +  alu_src1);
    `FUNCT_SUB     :  alu_dst_lo  =  (alu_src0 -  alu_src1);
    `FUNCT_MUL     : {alu_dst_hi,
                      alu_dst_lo} =  (alu_src0 *  alu_src1);
    `FUNCT_MULT    : {alu_dst_hi,
                      alu_dst_lo} =  (alu_src0 *  alu_src1);
    `FUNCT_AND     :  alu_dst_lo  =  (alu_src0 &  alu_src1);
    `FUNCT_OR      :  alu_dst_lo  =  (alu_src0 |  alu_src1);
    `FUNCT_XOR     :  alu_dst_lo  =  (alu_src0 ^  alu_src1);
    `FUNCT_NOR     :  alu_dst_lo  = ~(alu_src0 |  alu_src1);
    `FUNCT_SLLV    :  alu_dst_lo  =  (alu_src0 << alu_src1);
    `FUNCT_SRLV    :  alu_dst_lo  =  (alu_src0 << alu_src1);
    `FUNCT_MFHI    : ;
    `FUNCT_MFLO    : ;
    `FUNCT_MTHI    : ;
    `FUNCT_MTLO    : ;
    `FUNCT_JR      : ;
  endcase
end
endtask

task lw( input  [31:0] mem_addr,
         output [31:0] mem_data );
begin
  mem_data[ 7: 0] = MEM[{mem_addr[31:2],2'b00}];
  mem_data[15: 8] = MEM[{mem_addr[31:2],2'b01}];
  mem_data[23:16] = MEM[{mem_addr[31:2],2'b10}];
  mem_data[31:24] = MEM[{mem_addr[31:2],2'b11}];
end
endtask

task sw( input  [31:0] mem_addr,
         input  [31:0] mem_data );
begin
  MEM[{mem_addr[31:2],2'b00}] = mem_data[ 7: 0];
  MEM[{mem_addr[31:2],2'b01}] = mem_data[15: 8];
  MEM[{mem_addr[31:2],2'b10}] = mem_data[23:16];
  MEM[{mem_addr[31:2],2'b11}] = mem_data[31:24];
end
endtask
```

Listing 5.7 Verilog tasks to model the ALU and main memory.

the right point and updating `state` appropriately. Notice that we have broken the execute stage into execute, memory access and write to make things clearer; it then remains to implement the tasks that determine the processor behaviour within each state.

Before we implement these tasks, we introduce three auxiliary tasks in Listing 5.7 that are used to model the ALU and access to memory. The `alu` task simulates the ALU performing some operation, determined by `alu_oper`, on the source values `alu_src0` and `alu_src1` to produce the results `alu_dst_lo` and `alu_dst_hi`. The ALU behaviour is implemented using built-in Verilog op-

```
task fetch();
begin
  lw( PC, IR );
  PC = PC + 4;
end
endtask

task decode();
begin
  dec_opcode = IR[31:26];
  dec_rs     = IR[25:21];
  dec_rt     = IR[20:16];
  dec_rd     = IR[15:11];
  dec_funct  = IR[ 5: 0];

  dec_imm16  = { 16'h0000, IR[ 15:0]       };
  dec_imm26  = {  6'h00  , IR[ 25:0]       };
  dec_imm16s = { 14'h0000, IR[ 15:0], 2'h0 };
  dec_imm26s = {  4'h00  , IR[ 25:0], 2'h0 };

  dec_ext16  = { ( ( IR[15] == 1 ) ? 16'hFFFF :
                                     16'h0000 ), IR[15:0]       };
  dec_ext26  = { ( ( IR[25] == 1 ) ?  6'h3F   :
                                      6'h00   ), IR[25:0]       };
  dec_ext16s = { ( ( IR[15] == 1 ) ? 14'hFFFF :
                                     14'h0000 ), IR[15:0], 2'h0 };
  dec_ext26s = { ( ( IR[25] == 1 ) ?  4'h3F   :
                                      4'h00   ), IR[25:0], 2'h0 };
  GPR[0]     = 0;
end
endtask

task execute();
begin
  case( dec_opcode )
    `OPCODE_SPECIAL1 : alu( dec_funct, GPR[dec_rs], GPR[dec_rt]);
    `OPCODE_SPECIAL2 : alu( dec_funct, GPR[dec_rs], GPR[dec_rt]);
    `OPCODE_ADDI     : alu(`FUNCT_ADD, GPR[dec_rs], dec_ext16  );

    `OPCODE_LW       : alu(`FUNCT_ADD, GPR[dec_rs], dec_ext16  );
    `OPCODE_SW       : alu(`FUNCT_ADD, GPR[dec_rs], dec_ext16  );

    `OPCODE_BEQ      : alu(`FUNCT_ADD, PC,          dec_ext16s );
    `OPCODE_BNE      : alu(`FUNCT_ADD, PC,          dec_ext16s );
    `OPCODE_BLEZ     : alu(`FUNCT_ADD, PC,          dec_ext16s );
    `OPCODE_BGTZ     : alu(`FUNCT_ADD, PC,          dec_ext16s );
    `OPCODE_REGIMM   : alu(`FUNCT_ADD, PC,          dec_ext16s );

    `OPCODE_JAL      : alu(`FUNCT_ADD, PC,                   8  );
  endcase
end
endtask
```

Listing 5.8 Verilog tasks to perform fetch, decode and execute stages.

```
task memory();
begin
  case( dec_opcode )
    `OPCODE_LW : lw(alu_dst_lo, GPR[dec_rt]);
    `OPCODE_SW : sw(alu_dst_lo, GPR[dec_rt]);
  endcase
end
endtask

task write();
begin
  case( dec_opcode )
    `OPCODE_SPECIAL1 : case( dec_funct )
                         `FUNCT_SYSCALL : ;
                         `FUNCT_BREAK   : ;

                         `FUNCT_MULT : begin
                                         HI = alu_dst_hi;
                                         LO = alu_dst_lo;
                                       end
                         `FUNCT_MFHI : GPR[dec_rd] = HI;
                         `FUNCT_MFLO : GPR[dec_rd] = LO;
                         `FUNCT_MTHI : HI = GPR[dec_rs];
                         `FUNCT_MTLO : LO = GPR[dec_rs];

                         `FUNCT_JR   : PC = GPR[dec_rs];

                         default     : GPR[dec_rd] = alu_dst_lo;
                       endcase
    `OPCODE_SPECIAL2 : GPR[dec_rd] = alu_dst_lo;
    `OPCODE_ADDI     : GPR[dec_rt] = alu_dst_lo;

    `OPCODE_BEQ      : if( GPR[dec_rs] == GPR[dec_rt] )
                         PC = alu_dst_lo;
    `OPCODE_BNE      : if( GPR[dec_rs] != GPR[dec_rt] )
                         PC = alu_dst_lo;
    `OPCODE_BLEZ     : if( GPR[dec_rs] <= 0           )
                         PC = alu_dst_lo;
    `OPCODE_BGTZ     : if( GPR[dec_rs] >  0           )
                         PC = alu_dst_lo;

    `OPCODE_REGIMM   : case( dec_rt )
                         `RT_BLTZ : if( GPR[dec_rs] <  0 )
                                      PC = alu_dst_lo;
                         `RT_BGEZ : if( GPR[dec_rs] >= 0 )
                                      PC = alu_dst_lo;
                       endcase

    `OPCODE_J        : PC = dec_imm26s;
    `OPCODE_JAL      : begin
                         GPR[31] = alu_dst_lo;
                         PC      = dec_imm26s;
                       end
  endcase
end
endtask
```

Listing 5.9 Verilog tasks to perform memory access and write stages.

erators; in Chapter 7 we examine how this task can be replaced by a more efficient, more realistic module that does the same thing. The `lw` and `sw` tasks simulate memory access by loading and storing a 32-bit value to and from memory respectively. Again, in Chapter 8 we examine how this can be achieved more realistically by replacing the 2-dimensional `MEM` register with an accurate model with an appropriate interface.

Using the `alu`, `lw` and `sw` tasks as a means of abstraction, we can start to implement the processor behaviour. Listing 5.8 details the tasks used by the state machine for the fetch, decode and execute stages. Each performs exactly the behaviour we outlined in our previous description of execution within a stored program architecture. For example, the `fetch` task first loads the next instruction, using the address in *PC*, into the instruction register *IR*. It then uses the ALU to increment *PC*. The `decode` task decodes the instruction in *IR*, performing sign extension to produce values read to operate on. This task also cheats a little bit: to ensure that $GPR[0]$ always holds the value zero, we reset it here so that when we come to executing an instruction it is always right. The `execute` task performs the first stage of execution by invoking the ALU on source operands to produce useful results. In the case of an R-type arithmetic instruction, the ALU operator is stored in the `funct` field so we can easily work out what to do; in the case of a branch, the ALU is used to compute the branch target by adding the offset to *PC*. Notice how the general-purpose registers are indexed by the decoded register address. Finally, the memory access and write tasks are detailed in Listing 5.9. The `memory` task implements memory access, both load and store, simply using a call to the appropriate `lw` or `sw` task. In both cases, the address to access will have been computed during the execute stage and generated by the ALU into `alu_dst_lo`. The write task `write` deals with storing the results of execution back and also completing any branches whose target address was computed during the execution stage.

In order to test the model, we first need to present it with a program to execute. To initialise the memory with a program, as triggered by a positive edge on the reset signal, we use the `$readmemh` system task. This has the effect of reading the file specified as the first argument into the 2-dimensional register specified as the second argument at an offset specified by the third argument. In this case, the file `processor.bin` is filled with 256 lines which look like

```
01 00 03 20
02 00 04 20
20 10 64 00
00 00 00 00
.. .. .. ..
```

Given the bytes are presented in little-endian order, the first three 32-bit words represent the machine code instructions

$$00100000000000110000000000000001_{(2)}$$
$$00100000000001000000000000000010_{(2)}$$
$$00000000110010000010000000100000_{(2)}$$

which represent the very basic program

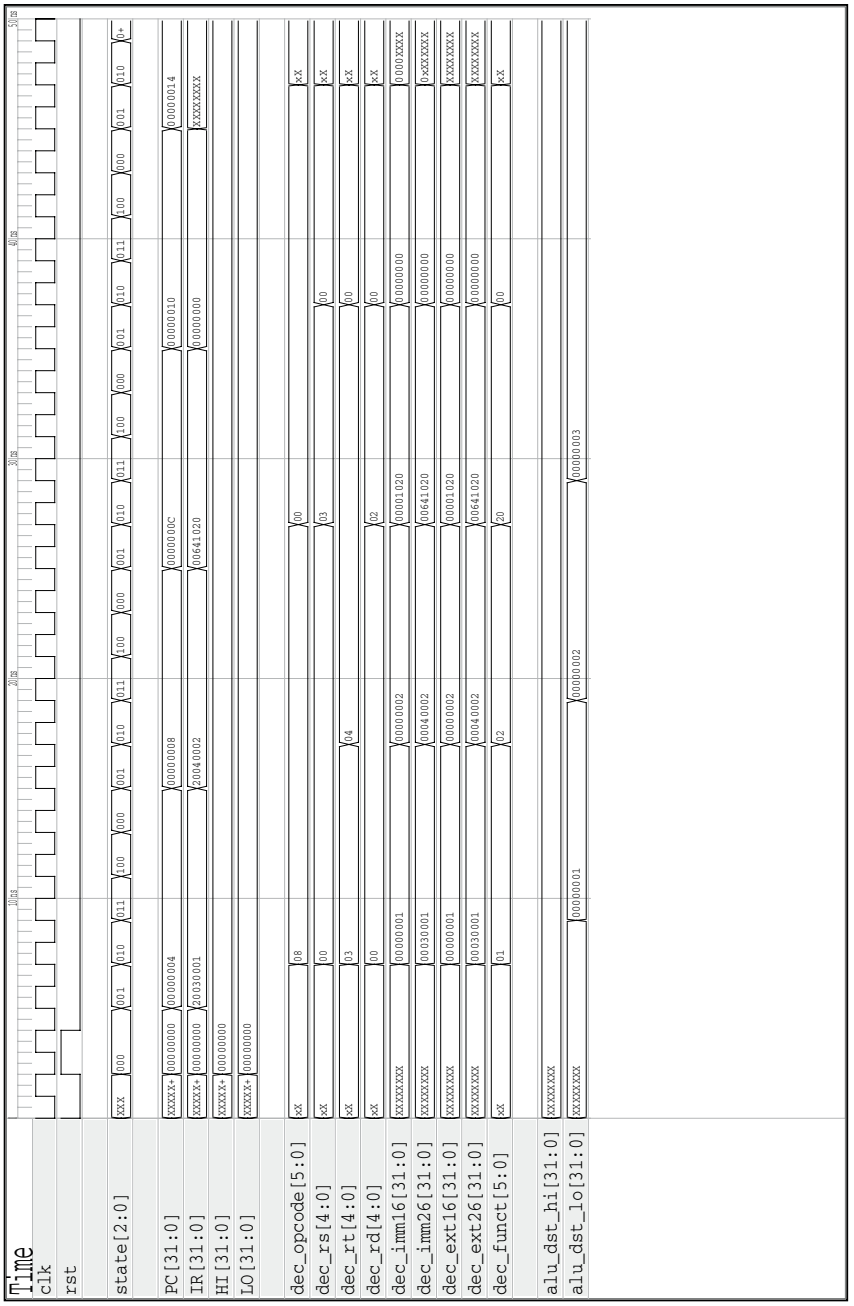


Figure 5.8 Behaviour of the processor model.

$$GPR[3] \leftarrow GPR[0] + 1$$
$$GPR[4] \leftarrow GPR[0] + 2$$
$$GPR[2] \leftarrow GPR[3] + GPR[4]$$

via two `addi` and one `add` instruction. Starting the processor executing on this program yields the behaviour in Figure 5.8. Iteration through the steps of execution is obvious with the `state` value being updated on clock edges once the processor is reset. One can see that during the fetch stage, i.e., when `state` equals 0, the *IR* value is read from memory and the *PC* is updated. On the first fetch this results in the instruction $00100000000000110000000000000001_{(2)} = 20030001_{(16)}$ being loaded from address $0_{(16)}$ and then *PC* being set to $4_{(16)}$. After parts of the instruction are extracted during the decode stage, the ALU is invoked to perform the operation $GPR[0] + 1$ which produces the result $1_{(16)}$ in `alu_dst_lo`. The value is written-back and reused later as execution continues.

5.8 Further Reading

- S.P. Dandamudi.
  Guide to RISC Processors.
  Springer-Verlag, 2005. ISBN: 0-387-21017-2.
- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.
- J.L. Hennessy and D.A. Patterson.
  Computer Architecture: A Quantitative Approach.
  Morgan-Kaufmann, 2002. ISBN: 1-558-60724-2.
- D.A. Patterson and J.L. Hennessy.
  Computer Organization and Design: The Hardware/Software Interface.
  Morgan-Kaufmann, 2004. ISBN: 1-558-60604-1.
- J. Stokes.
  Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture.
  No Starch Press, 2006. ISBN: 1-593-27104-2.
- D. Sweetman.
  See MIPS Run.
  Morgan-Kaufmann, 2006. ISBN: 0-120-88421-6.
- A.S. Tanenbaum.
  Structured Computer Organisation.
  Prentice-Hall, 2005. ISBN: 0-131-48521-0.

5.9 Example Questions

21. Imagine you are on the design team for a new 16-bit processor. The ISA for the processor includes specifications for:

- 16 general-purpose registers.
- 8 ALU instructions that take two input register operands and write a result into a third output register operand.
- 4 branch instructions that take an input register operand and an 8-bit immediate offset.
- 2 memory access instructions that take two input register operands and one output register operand. 2 memory access instructions that take one input register operand, one output register operand and a 4-bit immediate offset.

a. Devise an instruction encoding for the processor, taking care to explain any advantages and disadvantages of your design.
b. Draw a block diagram of the basic components within this processor, explain the function of each one and how they are connected together.
c. Using the example of an instruction that adds two registers together and writes the result in a third register, describe the steps involved during the following phases of execution:

   i. Fetch.
   ii. Decode.
   iii. Execute.

d. The control unit that operates the data-path can be implemented using hardwired logic or a micro-code based system. Explain the idea of both of these methods, including the advantages and disadvantages of each.

22. Explain what is meant by the following addressing modes, giving examples of their use in a C program:

a. Immediate.
b. Direct.
c. Indirect.
d. Indexed.

23. a. A change in the design of a particular processor means it will now use the idea of predicated execution. Explain what this means and translate the following C fragment into an assembly language style program for the processor:

```
t = 0;

for( int i = 0; i < n; i++ ) {
  if( ( x >> i ) & 1 )
    t = t + 1;
}
```

Use predicated execution or branch instructions where appropriate and take care to justify your choice in each case.

b. It is suggested that the code density of programs executed by the processor is too low. Explain what is meant by code density and outline **two** approaches to improving this situation.

24. You have been asked to define the instruction set of a new processor and the encoding of this instruction set. The processor contains a program counter (PC), four general-purpose registers (A, B, C and D), uses 8-bit instructions and has a 256-byte memory that is addressed indirectly via the registers. The instructions set includes the following instructions:

Name	Operands	Meaning
LDA	x	Loads constant x, with $0 \leq x \leq 127$, into register A.
MOV	R, S	Moves register S into register R.
NOT	R	Performs a bitwise complement of register R.
NOP		Performs no operation.
NEG	R	Performs twos-complement negation of register R.
BEZ	R, x	Branches by incrementing the program counter by an offset x, with $-8 \leq x \leq 7$, if register R is equal to 0.

So one might write the program:

```
LDA 10
MOV B, A
NEG B
BEZ B, 4
```

to load the constant 10 into register A, then move this value into register B and finally negate it before jumping forward by four instructions if register B turns out to be zero.

a. Design an encoding for the instructions given above.
b. What is the maximum number of 2-operand instructions that you can add to the instruction set ?
c. What is the maximum number of 1-operand instructions that you can add to the instruction set ?
d. What is the maximum number of 0-operand instructions that you can add to the instruction set ?
e. Write down a more complete instruction set, adding those which you view as currently missing and giving reasons why you want to add them.
f. Finally, write down the instruction encoding for this complete instruction set.

25. There are 32 pieces on a chess board represented by an 8×8 grid; a black and a white version of 1 king, 1 queen, 2 bishops, 2 knights, 2 rooks and 8 pawns.

You are involved in the design of a robot that automates the game of chess **only** in terms of moving the physical pieces around. The robot needs to be fed a list of

moves so that it knows how to move a piece on one board position to another position; it does not care about the legality of moves. However, it is intelligent enough to know whose turn it is (i.e., black or white), how to find a piece on the board given its unique name (e.g., pawn #1) and can cope with capturing pieces if a move places a piece on an already occupied square. It cannot play any version of chess with the advanced rules of promotion, castling, resignation or en passant: it simply moves and captures pieces. It can deal with the fact that pawns can move two spaces forward from their starting position but otherwise has no memory.

Design a compact encoding of instructions that can be fed to the robot to provoke the movement of any piece on the board. State any assumptions you make in your design and the advantages and disadvantages of the result.

26. Imagine you are asked to design a RISC style instruction set architecture for a new 32-bit processor which is modelled roughly on the MIPS32 design. The processor has 32 general-purpose registers and the instruction set is to include:

- Arithmetic, logic and comparison instructions that use a mix of general purpose register and immediate operands.
- Memory access instructions that use an single addressing mode whereby a base address is specified by a general-purpose register operand and an offset is specified by an immediate operand.
- Branch instructions that include both conditional and unconditional variants and use mix of general-purpose register and immediate operands to specify the absolute and offset target address.

a. One designer wants to use a fixed length instruction encoding while another suggests that a variable length encoding would be better.

   i. Explain the advantages and disadvantages of each approach.
   ii. Select **one** approach and using your selection, design basic instruction encoding formats for the instruction types listed above.

b. The "add immediate" instruction adds the contents of a general-purpose register to a 16-bit immediate and stores the result back into a general-purpose register. For example, the instruction $GPR[1] \leftarrow GPR[2] + 3$ takes the contents of general-purpose register 2, adds the immediate value 3 to it and stores the result into general-purpose register 1.
   Using just the instruction encoding formats from above, show how

   i. $GPR[4] \leftarrow GPR[8] + 10$
   ii. $GPR[4] \leftarrow GPR[8] - 10$

   are encoded into 32-bit "add immediate" instructions; state any assumptions you make.

c. A major customer for the processor has identified the following C function as crucial to the performance of their applications:

```
int f( int x ) {
  int w = 0;

  for( int i = 0; i < 32; i++ ) {
    if( ( x >> i ) & 1 )
      w = w + 1;
  }

  return w;
}
```

Explain what this function does and how you might alter the processor design (including any changes to the ALU) to satisfy the requirements of this customer.

d. There is some debate about which endian convention the processor should use. Assuming the memory is byte addressed, consider the execution of an instruction that stores the 32-bit integer 305419896 at address 0. List the content of these addresses after the store instruction completes if the processor uses

   i. Little-endian byte ordering.
   ii. Big-endian byte ordering.

e. The team responsible for writing a compiler for the new processor is concerned about the number of general-purpose registers. Explain the positive and negative implications for the instruction set if the number of general-purpose registers is

   i. Increased from 32 to 64.
   ii. Decreased from 32 to 16.

   In **each** case, describe how this might alter performance of programs that execute on the processor.

Chapter 6
Measuring Performance

One accurate measurement is worth a thousand expert opinions.

– G. Hopper

He uses statistics as a drunken man uses lampposts: for support rather than illumination.

– A. Lang

Abstract Previously we developed a simple processor data-path; our motivation was simply to demonstrate the fundamentals of how such a processor might work. We can measure the performance of such a processor according to a huge number of criteria. For example, we might consider how many instructions it executes in a given time period, how much power it uses, how large the manufactured circuit is, and so on. In a more real-life context, we might tune our design to give the best performance according to one of these criteria; in this chapter we investigate a number of ways to measure performance (focusing on the speed at which a processor executes programs).

6.1 Measuring Performance

Before we start to examine better processor designs we need to be able to evaluate their relative **performance** characteristics, i.e., understand what "better" means. The problem is that performance can be a difficult concept to define exactly. It is often more instructive to think about the overall **quality** of a design rather than performance. Quality can combine lots of different features as dictated by different application or market areas; a **cost function** will determine how quality is evaluated by combining many **metrics**. For example, one might worry about how quickly a processor executes a given program, or how much power is used during execution. In most designs, such metrics compete for resources in the sense that no one can be satisfied without being detrimental to another; some form of trade-off or compro-

mise must be reached, one can think of this as maximising the cost function in some way.

For a general circuit, perhaps the most simple metric one can consider is how quickly it operates. This can be further decomposed into two standard measures.

Definition 32. The **latency** of a circuit is the total time elapsed before a given input is operated on to produce an output. The **bandwidth** or **throughput** of a circuit is the rate at which new inputs can be offered (or outputs produced).

The difference is somewhat subtle but we will see why *both* are important when we look at improving performance: for now, simply consider latency. From the basic definition, one can try to capture what is meant by performance.

Definition 33. The **performance** of design X is inversely proportional to latency

$$\textsc{Performance}(X) = \frac{1}{\textsc{Latency}(X)}.$$

From this, we can define what is meant by the statement "design X is n times faster than design Y". That is, if n is the speed-up offered by design X over design Y, then

$$\begin{aligned}\textsc{Speed-Up}(X \text{ over } Y) &= \frac{\textsc{Latency}(Y)}{\textsc{Latency}(X)} \\ &= \frac{\textsc{Performance}(X)}{\textsc{Performance}(Y)} \\ &= n.\end{aligned}$$

In the context of a processor rather than a general circuit, one typically uses **execution time** in place of latency; that is, the time taken to execute a given program. However, when placed in our definition this presents even more problems: what do we mean by execution time, how do we measure time and what *exactly* are we executing ?

6.1.1 Estimating Execution Time

There are plenty of bad ways to evaluate the performance of a processor that are none-the-less still used in marketing literature. A prime example is the **Millions of Instructions Per Second (MIPS)** metric, a different MIPS from the MIPS32 ISA ! Essentially the MIPS metric measures how many instructions a processor can execute each second: the assertion is that the more instructions executed each second, the higher the performance. One might therefore try to estimate the execution time of program X on processor Y using

$$\textsc{Execution-Time}(X,Y) = \frac{\textsc{Instruction-Count}(X)}{\text{MIPS-}\textsc{Rating}(Y)}.$$

This is an intuitive, easy to understand measure with which one can compare the performance of processors. In fact, in the 1970s the performance of computers was quoted relative to a baseline computer; the VAX 11/780 was said to represent 1 VAX MIPS, the VAX executed one million instructions per-second. The problem is, this method of comparison is quite flawed. In particular, it should be clear that a RISC processor will typically take many instruction to execute a given program since each instruction performs a relatively simple operation; a CISC processor will typically take less instructions since each instruction performs more complex operations. Comparing one with the other is nonsense: we need a performance metric that includes the idea of amount of useful work done. That is, a RISC processor will almost always have a higher MIPS rating than a CISC alternative; regardless of the design quality. In addition, the MIPS rating is tied to the clock speed of the physical processor. In theory at least, it would be nice to evaluate the quality of a design independently from the clock speed.

The **Cycles Per-Instruction (CPI)** metric hopes to solve this problem to some extent. It measures the average number of clock cycles needed per-instruction and varies according to the processor architecture and the program being executed. The **instruction mix** for a processor tells us what ratio of different instruction types are used in the average program. For example, we might note that 40% of instructions perform arithmetic, 40% perform memory access and 20% perform branches. The process of discovering the instruction mix for a processor is called **workload characterisation**; we typically use a range of benchmark programs to produce an overall picture of how often different instructions are used.

Given the instruction mix and the CPI for each instruction type, we can estimate the overall CPI for the processor. The CPI for each instruction is easily identified; typically they are quoted as part of the processor manual. Let CPI_i denote the CPI for instruction type i and F_i denote how often instruction type i is executed. The overall CPI is then calculated as

$$\mathrm{CPI} = \sum_{i=0}^{n-1} \mathrm{F}_i \cdot \mathrm{CPI}_i.$$

Consider an example instruction mix:

Operation	F_i	CPI_i	$\mathrm{F}_i \cdot \mathrm{CPI}_i$
Arithmetic	40%	1	0.4
Memory	40%	4	1.6
Branch	20%	2	0.4
Total	100%		2.4

Using the overall CPI, we can then estimate how long a given program will take to execute. Firstly we can estimate the number of cycles the program will take to execute; this requires we know the number of instructions executed by the program and the overall CPI for the processor as computed above. Then, for program X on processor Y we have

$$\text{CYCLES}(X,Y) = \text{INSTRUCTION-COUNT}(X) \cdot \text{CPI}(Y).$$

Finally we can estimate the execution time by dividing the number of cycles by the clock rate:

$$\begin{aligned}\text{EXECUTION-TIME}(X,Y) &= \frac{\text{CYCLES}(X)}{\text{CLOCK-RATE}(Y)} \\ &= \frac{\text{INSTRUCTION-COUNT}(X) \cdot \text{CPI}(Y)}{\text{CLOCK-RATE}(Y)}\end{aligned}$$

6.1.2 Measuring Execution Time

Rather than estimating execution time of a given program, one might simply measure how long it takes to execute. Typically we use the term **wall-clock time** to mean the elapsed time for execution. However, this may include components other than the workload under examination such as time to load the program, time spent waiting for access to resources such as storage devices and so on. Depending on what we want to get from the exercise, we might include or exclude these components. Time is then one of:

Wall-clock Time Also known as execution time or response time, this is the total time required to execute a program, including all the potential overheads.

Processor Time Unlike wall-clock time which includes overheads, processor time is the total computational time spent during execution i.e., without overheads. That is, processor time is the amount of time the program spends having instructions executed by the processor.

System Time The amount of time spent executing operating system or library code on behalf of the program under examination. This might include calls to functions that load data from a storage device for example.

User Time In contrast with system time, user time measures how much time is spent executing the program itself.

Most UNIX systems include the `time` command which is able to report wall-clock, system and user times for the execution of a given program. The resolution (or accuracy) of the results is not ideal, but it offers an easy first attempt at measuring how long a program takes to execute. On a UNIX system for example, we might use the `wc` program to count the number of words in a local dictionary named `/usr/share/dict/words`; with the `time` command we can measure how long this takes using the following:

```
bash$ time wc /usr/share/dict/words
 483523  483523 4992010 /usr/share/dict/words

real    0m0.714s
user    0m0.656s
sys     0m0.011s
```

```
#include <time.h>

int main( int argc, char* argv[] )
{
  clock_t before;
  clock_t after;
  clock_t total;

  before = clock();
  ...
  after  = clock();

  total  = ( after - before ) / CLOCKS_PER_SEC;
}
```

Listing 6.1 Use of `clock` to measure execution time.

```
unsigned uint64 rdtsc()
{
  register uint32 ax;
  register uint32 dx;

  __asm__ __volatile__( "rdtsc" : "=a"( ax ),
                                  "=d"( dx ) );

  return ( ( uint64 )( ax ) <<  0 ) |
         ( ( uint64 )( dx ) << 32 ) ;
}

int main( int argc, char* argv[] )
{
  uint32 before;
  uint32  after;
  uint32  total;

  before = rdtsc();
  ...
  after  = rdtsc();

  total  = ( after - before );
}
```

Listing 6.2 Use of `rdtsc` to measure execution time.

We can see that the total time taken was 0.714 seconds, of which only about 0.656 was spent executing the actual `wc` program.

Listing 6.1 and Listing 6.2 demonstrate two common methods of measuring execution time of a program fragment with a higher degree of accuracy; in both cases the continuation dots represent the fragment being timed. Listing 6.1 uses the C `clock` function which returns an approximation of how much time has elapsed since the beginning of execution. It has roughly a microsecond accuracy. By recording the value before the fragment and afterwards, the total time taken is the difference between the two. It is common to divide by `CLOCKS_PER_SEC`, a processor-specific constant that allows conversion into a number of seconds. Deviating for a moment from our use of MIPS32 as an example, Listing 6.2 is specific to Intel

processors. It uses some inline assembly language to access the `rdtsc` instruction that returns the **Time Stamp Counter (TSC)** value. The TSC is a register internal to the processor, an example of a machine-specific register. Every time a processor cycle is completed, the register is incremented; the program reading the TSC can thus measure time, very precisely, in number of processor cycles.

6.1.3 Benchmark Programs

Although we now know how to measure or estimate the execution time for a given program on a given processor, another problem remains. Specifically, it is not clear *which* program we should be measuring (and what input data to provide) so we get a meaningful result. Clearly these choices will have a major effect on our analysis. For example, a program that performs mostly memory accesses will probably perform differently than a program that performs a lot of computation; input data that is hard to process or causes worst-case behaviour in the constituent algorithms will provoke worse performance than more friendly input. As such, to get an idea of processor performance that is independent of the program being executed, we need to make choices that match typical usage. That is, since we cannot measure the execution time of all possible programs using all possible inputs, we need to find a representative sample of programs (and input data) that can be easily extrapolated to how the processor will really be used.

A **benchmark** is a program or set of programs selected and used to act as the required sample. Benchmarks allow one to perform comparisons between processor designs by simply measuring and comparing execution time, and to extrapolate to other programs by performing workload characterisation and deriving typical instruction mixes. A benchmark should, in general, be representative of the type of applications run on the processor, not be overly dependent on specifics of the processor (unless one wants to test a specific feature), and be reproducible in the sense that we compare and verify results. Benchmarks can usually be placed in one of three main classes:

Application A benchmark application is a single program, specifically selected, or even specifically written, to test particular aspects of performance. Examples include 3DMark series of benchmarks which are used to test the performance of 3D graphics rendering in modern graphics cards. Using a single application enables one to focus on specific items of interest but the results are often not easy to extrapolate because of this.

Suite A benchmark suite is typically a collection of benchmark applications. Where a single application tests only one aspect of performance, the suite captures many aspects by constructing some form of average over all the component applications. This gives less specific results but makes it easier to see trends that apply in general. Examples include the SPEC benchmarks, original versions of which included commonly used UNIX programs such as `gzip` and `ghostview`.

Kernel A benchmark kernel is a very small fragment of program; it might not do anything useful alone, but represents the typical computational core of the programs we are interested in. Kernels are usually easy to program and analyse but do not usually test overall system performance. Examples include the Livermore Loops [41] a set of Fortran, and later C, kernels that compute common arithmetic functions such as matrix multiplication and vector dot product. They focus mainly on program fragments that loop over arrays of data reasoning that this is the main computational core of scientific applications.

6.1.4 Measuring Improvement

Our initial definition of the speed-up given by design X over design Y is somewhat naive. To be more accurate, we need to consider **Amdahl's Law** which, roughly speaking, states that the performance improvement of using some faster mode of execution is limited by the fraction of the time the faster mode can be used. In short, Amdahl's Law is just a slightly technical way of stating the well-known law of diminishing returns. That is, if an improvement implemented in design X is not used very often the benefit in performance will be small: the less it is used, the smaller the improvement will be.

Imagine we improve some hardware design (perhaps some part of the ALU) H within in a basic processor design Y to produce a new processor design X. Also, suppose the design for H in processor X speeds-up the original design in processor Y by some factor F. Finally, imagine we have estimated that the proportion of overall time that either processor spends using the device H is P. The new execution time is derived as

$$\text{EXECUTION-TIME}(X) = \text{EXECUTION-TIME}(Y) \cdot ((1-P) + (P/F)).$$

We can also work out the speed-up the improvement produced

$$\begin{aligned}\text{SPEED-UP}(X \text{ over } Y) &= \frac{\text{EXECUTION-TIME}(Y)}{\text{EXECUTION-TIME}(X)} \\ &= \frac{1}{(1-P)+(P/F)}.\end{aligned}$$

To see that this is correct, consider the denominator of the fraction. The overall execution time of Y will be the execution time of the unimproved portion of the design, which is $1-P$, plus the execution time of the improved portion, which is P/F since we improved it by the factor F. Finally, we divided the old execution time by the new execution time just like in our original formula.

Consider an example; imagine processor Y takes ten seconds to execute a program. Then processor X is designed to improve on Y by using a new adder design, maybe it uses a carry look-ahead adder. The adder is used in about 20% of instructions and the new adder is four times as fast. Thus, we compute

$$\begin{aligned}\textsc{Speed-Up}(X \text{ over } Y) &= \frac{1}{(1-P)+(P/F)} \\ &= \frac{1}{(1-0.2)+(0.2/4)} \\ &\sim 1.18.\end{aligned}$$

This might come as some surprise; our adder is four times as fast so why is the whole system not *much* faster ?! This is the law of diminishing returns in effect: unless the adder is used more often, the improvement will not be capitalised on. As it is, processor X will take ~ 8.47 seconds using the new adder.

6.2 Further Reading

- L. Eeckhout, L.K. John and R.P. Wilson.
  Performance Evaluation and Benchmarking.
  CRC Press, 2005. ISBN: 0-849-33622-8.
- R.W. Hockney.
  The Science of Computer Benchmarking.
  Society for Industrial & Applied Mathematics, 1996. ISBN: 0-898-71363-3.
- R. Jain.
  The Art of Computer Systems Performance Analysis.
  John Wiley, 1991. ISBN: 0-471-50336-3.

6.3 Example Questions

27. a. CPI and MIPS are both measures of processor performance. Explain how they are calculated and their advantages and disadvantages.

b. After some experimentation, someone shows that programs written for a given processor have the following instruction mix:

	Frequency	Cycles
ALU	50%	1
Load	20%	5
Store	10%	3
Branch	20%	2

The addition of a new addressing mode and a better compiler means there are less accesses to memory required. The instruction mix after this change is

	Frequency	Cycles
ALU	55%	1
Load	10%	5
Store	10%	3
Branch	25%	2

Calculate the overall CPI for programs that run on the processor before **and** after the change, and the speed-up that results from the change.

28. Outline the difficulties faced when writing benchmark software that is to collect information about the processor performance. Include at least one method that could be used to artificially increase the performance, i.e., "cheat" the benchmark.

Chapter 7
Arithmetic and Logic

The whole of arithmetic now appeared within the grasp of mechanism.

– C. Babbage

Abstract So far we have seen how to represent numbers as binary vectors and how, using transistors, one can implement logic gates that perform the basic Boolean operations NOT, AND, and OR. In order to realise the goal of performing useful computation on integer values represented as n-bit vectors, we need to compose these logic gates into higher-level circuits. Generally speaking, the **Arithmetic and Logic Unit (ALU)** packages circuits for different operations into one component that performs computation for the processor. To aid the discussion in examples we set $n = 8$ throughout, i.e., set the size of a word to equal one byte. However, it is important to see that the methods are presented in a general form: one can easily extend them to cope with larger word sizes.

7.1 Introduction

The ALU is at the heart of any processor; when one requires an addition or comparison to happen, *it* is the component that performs the computation under the guidance of a control unit. This design was first formalised by John von Neumann in 1945 in his pioneering design for the EDVAC computer. Many principles outlined in his design are still retained today. Specifically he noted that any computer would have to perform basic mathematical operations on numbers and that it is "reasonable that [the computer] should contain specialised organs for these operations" [48]. The modern ALU is an example of such an organ.

In a very rough sense one can view the ALU as a primitive calculator: it accepts operands as input, waits to be told what operation to perform, and then produces results as output after some time. As such, one can view the high-level architecture of a basic ALU as being similar to Figure 7.1. In this diagram, there are a number of units within the ALU which all take inputs and generate results, potentially in

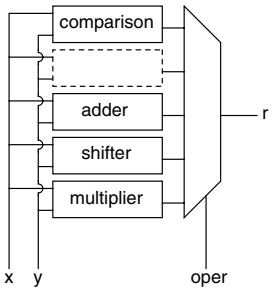


Figure 7.1 A high-level architectural design for a basic ALU.

parallel, for their associated operations. For example, we might have one unit for addition, one unit for multiplication and one unit for shift operations. These results are selected between by a signal which is determined by the control-path; depending on which operator it uses, a different result is produced on the ALU output. In reality it might be advantageous to turn off units which are not being used at any one time so as to reduce power consumption or heat. Additionally it is very likely that replicated logic in different units could be shared; for example, the unit for addition might also be used in multiplication. We will not worry about these sorts of optimisation here. Finally note that at this high-level, one usually views the ALU as a single component. However, at a lower-level, it might be attractive to split it into parts. For example, it is common to have dedicated floating point arithmetic within a **Floating Point Unit (FPU)**, or to split the integer ALU into two parts for arithmetic and comparisons. In this way, one can perform different types of computation at the same time; for example, we might compare two numbers and add another two numbers at the same time.

7.2 Comparisons

Recall from Chapter 2 that we can easily construct functions that describe 1-bit equality and less than comparisons; we use $equ(x,y)$ to denote the truth value related to the test $x = y$, and $lth(x,y)$ to denote the truth value related to $x < y$ for 1-bit values x and y. The basic idea in constructing comparators for larger values, n-bit integers in our case, is to combine many of these 1-bit comparators together in order to produce the final result.

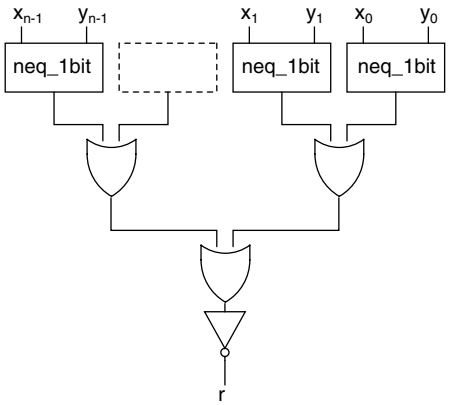


Figure 7.2 Design for an unsigned equality comparator.

```
1  module equ_nbit( r, x, y );
2
3    parameter n = 8;
4
5    output wire         r;
6    input  wire [n-1:0] x;
7    input  wire [n-1:0] y;
8
9           wire [n-1:0] w0 = x ^ y;
10
11   assign r = ~( |w0 );
12
13 endmodule
```

Listing 7.1 Implementation of an n-bit unsigned equality comparator.

7.2.1 Unsigned Comparisons

7.2.1.1 Equality

Implementing a hardware circuit to test values for equality is a simple matter of using the 1-bit equality test in a component-wise manner on the inputs and then ensuring all the 1-bit comparisons are true. This final step can be implemented by ANDing all the 1-bit comparisons together; the whole values are unequal if even one of the 1-bit comparisons fail, so all must be true for the comparison to be true. For efficiency, one can arrange this operation as a tree to minimise the critical path.

However, we can be slightly more clever than this naive approach and reduce the number of required gates somewhat. Instead of 1-bit equality modules we use 1-bit not-equal modules, which are essentially just XOR gates; this change saves n NOT gates. By ORing these results together, instead of ANDing, and then negating the result we compute the same function: we have just rearranged the expression a little.

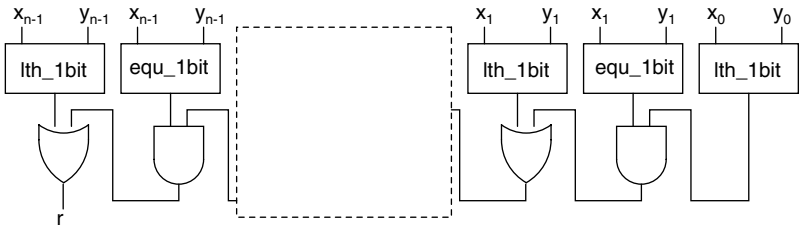


Figure 7.3 Design for an unsigned less than comparator.

```
1  module lth_nbit( r, x, y );
2
3    parameter n = 8;
4
5    output wire         r;
6    input  wire [n-1:0] x;
7    input  wire [n-1:0] y;
8
9           wire [n-1:0] w0 = ~(  x ^ y );
10          wire [n-1:0] w1 =  ( ~x & y );
11          wire [n-1:0] w2;
12
13   assign w2[0] = w1[  0];
14   assign r     = w2[n-1];
15
16   genvar i;
17   generate
18   begin
19     for( i = 1; i < n; i = i + 1 )
20     begin:gen_lth
21       assign w2[i] = w1[i] | ( w0[i] & w2[i-1] );
22     end
23   end
24   endgenerate
25
26 endmodule
```

Listing 7.2 Implementation of an n-bit unsigned less than comparator.

The end result is shown in Figure 7.2 and implemented in Listing 7.1, the overall saving is $n-1$ NOT gates. Note that the implementation uses a reduction operator: the assumption is that a Verilog tool-chain will be clever enough to translate this into a tree structure.

7.2.1.2 Less Than

To test if one n-bit value x is less than another one called y, we compare the i-th bits of x and y with each other starting from the most-significant and find that $x < y$ if either of the following is true:

1. $lth(x_i, y_i)$.
2. $equ(x_i, y_i)$ and taking x' and y' to be the $(i-1)$-th down to the 0-th bits of x and y respectively, we recursively find that $x' < y'$.

This is somewhat of an awkward description; more concretely we can write the operation on 8-bit values as the sequence

$$\begin{aligned}
t_0 &= lth(x_0, y_0) \\
&\vdots \\
t_5 &= lth(x_5, y_5) \vee (equ(x_5, y_5) \wedge t_4) \\
t_6 &= lth(x_6, y_6) \vee (equ(x_6, y_6) \wedge t_5) \\
t_7 &= lth(x_7, y_7) \vee (equ(x_7, y_7) \wedge t_6)
\end{aligned}$$

with the final result appearing as t_7. Consider an even smaller example where we set $x = 6_{(10)} = 110_{(2)}$ and $y = 7_{(10)} = 111_{(2)}$ such that $n = 3$. That is, we want to test if $110_{(2)} < 111_{(2)}$. Examining each bit of x and y we can clearly see that

$$\begin{aligned}
lth(x_0, y_0) &= \text{true} \\
lth(x_1, y_1) &= \text{false} \\
lth(x_2, y_2) &= \text{false}
\end{aligned}$$

and also that

$$\begin{aligned}
t_0 &= \text{true} \\
t_1 &= \text{true} \\
t_2 &= \text{true}
\end{aligned}$$

because

$$\begin{aligned}
t_0 &= lth(x_0, y_0) \\
&= \text{true}
\end{aligned}$$

$$\begin{aligned}
t_1 &= lth(x_1, y_1) \vee (equ(x_1, y_1) \wedge t_0) \\
&= \text{false} \vee \text{true} \\
&= \text{true}
\end{aligned}$$

$$\begin{aligned}
t_2 &= lth(x_2, y_2) \vee (equ(x_2, y_2) \wedge t_1) \\
&= \text{false} \vee \text{true} \\
&= \text{true}
\end{aligned}$$

Thus we find that $x < y$, or rather than $6_{(10)} < 7_{(10)}$, because the final output in the sequence is $t_2 = \text{true}$. Hopefully it is clear that we can chain together instances of our 1-bit comparator modules in order to implement the sequence above; this approach is shown in Figure 7.3 and implemented in Listing 7.2. Wires `w0` and `w1` implement the 1-bit equality and less than tests respectively; the generate statement chains these together as described above to produce the result `r`.

7.2.2 Signed Comparisons

A signed equality comparison is the same as an unsigned equality comparison since we are essentially just testing corresponding bits against each other. Signed less than comparison is a little more tricky however. We can formulate a set of rules for how signed less than behaves depending on the signs of the inputs x and y:

$$\begin{aligned}
&x \text{ +ve and } y \text{ -ve } \rightarrow x \not< y \\
&x \text{ -ve and } y \text{ +ve } \rightarrow x < y \\
&x \text{ +ve and } y \text{ +ve } \rightarrow x < y \text{ if } \textsc{Abs}(x) < \textsc{Abs}(y) \\
&x \text{ -ve and } y \text{ -ve } \rightarrow x < y \text{ if } \textsc{Abs}(y) < \textsc{Abs}(x) \,.
\end{aligned}$$

Thus we can realise a signed less than comparison just using some simple tests of the MSBs of x and y, which determine their sign, and controlled use of our unsigned less than comparison circuit.

The final thing to note is that once we have built hardware for equality and less than comparison, other operations can be derived using low-cost identities rather than additional dedicated circuits. For example, one can easily verify that

$$\begin{aligned}
&x \neq y \rightarrow \neg(x = y) \\
&x \leq y \rightarrow (x < y) \vee (x = y) \\
&x \geq y \rightarrow \neg(x < y) \\
&x > y \rightarrow \neg(x < y) \wedge \neg(x = y)
\end{aligned}$$

such that all six useful comparisons can be generated with hardware to test for equality and less than, and four extra logic gates.

7.3 Addition and Subtraction

7.3.1 Addition

Most people learn the method for multi-digit addition in school; we describe the steps in Algorithm 7.1. Say we select example inputs $x = 123_{(10)}$ and $y = 218_{(10)}$. Using our algorithm, we can produce a trace which details the values before and after major assignments as execution of the algorithm progresses:

i	x_i	y_i	c	t	z_i	c
0	3	8	0	11	1	1
1	2	1	1	4	4	0
2	1	2	0	3	3	0

and find that the result is $z = 341_{(10)}$. However, we are often used to writing such operations more diagrammatically. To calculate the sum $z = x + y$ in base b by hand for example, we first write out the digits of x and y with similar powers of b above

Input: The n-digit integers x and y in a base-b representation.
Output: The $n+1$-digit integer $z = x + y$ in a base-b representation.

```
1  ADD(x, y)
2      z ← 0
3      c ← 0
4      for i = 0 upto n − 1 step 1 do
5          t ← x_i + y_i + c
6          z_i ← t mod b
7          c ← ⌊t/b⌋
8      z_n ← c
9      return z
10 end
```

Algorithm 7.1: An algorithm for multi-digit addition.

each other. In decimal, this should be a familiar process; again taking $x = 123_{(10)}$ and $y = 218_{(10)}$ we produce the sum

$$\begin{array}{r} 1\ 2\ 3 \text{ input } x \\ \underline{2\ 1\ 8} \text{ input } y \\ \underline{0\ 1\ 0} \text{ carry} \\ \underline{3\ 4\ 1} \text{ result } z = x + y \end{array}$$

Note the zero in the first carry-in denoting that there is implicitly no carry into the first addition: we count this as zero. The same algorithm or diagrammatic method works in any base we care to represent our numbers in. If we consider binary numbers $x = 123_{(10)} = 1111011_{(2)}$ and $y = 281_{(10)} = 11011010_{(2)}$, we proceed as follows:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 0\ 1\ 1 \text{ input } x \\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 0 \text{ input } y \\ \underline{1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0} \text{ carry} \\ \underline{1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1} \text{ result } z = x + y \end{array}$$

to again compute the result $z = x + y = 341_{(10)} = 101010101_{(2)}$. Notice the carries propagate from right to left: we produce a carry from the i-th step which is propagated into the sum for the $(i+1)$-th step.

Although our algorithm works fine in general, we are presented with a problem when restricted to working with n-bit integers. The problem, termed overflow, is that the result of adding two n-bit integers together can create an $(n+1)$-bit result which is obviously too large to represent in n bits. This occurs when there is a carry-out of step $n-1$ of our algorithm and is demonstrated perfectly by the example above. Specifically, the inputs $x = 123_{(10)} = 1111011_{(2)}$ and $y = 281_{(10)} = 11011010_{(2)}$

are both 8 bits long but the result $z = x + y = 341_{(10)} = 101010101_{(2)}$ is 9 bits long. Given we want to represent z in 8 bits we have two main choices

- **Truncate** the result to 8 bits such that $z = 85_{(10)} = 01010101_{(2)}$ and signal by some method that an overflow has occurred so the error can be detected.
- **Clamp** or saturate the result to the highest value representable in the 8-bit size so that we have that $z = 255_{(10)} = 11111111_{(2)}$.

Clamped addition is useful in some contexts where accuracy is not important, for example when computing with graphical information represented as pixels. However, for other contexts we prefer to simply truncate the result and flag that overflow has occurred.

This detection of overflow is slightly complicated when one considers signed numbers. As an example, consider adding together $x = 127_{(10)}$ and $y = -127_{(10)}$ represented as 8-bit twos-complement binary vectors:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \text{ input } x \\ \underline{1\ 0\ 0\ 0\ 0\ 0\ 0\ 1 \text{ input } y} \\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0 \text{ carry} \\ \underline{1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \text{ result } z = x + y} \end{array}$$

We truncate the result to 8 bits and find $z = 0_{(10)} = 0_{(2)}$ but there is a carry-out of the last step which means $z_9 = 1$. Using the reasoning above, this should mean there is a carry-out yet we know the result should be zero and this can clearly be represented as an 8-bit value so what went wrong ? In short, we need to consider the sign of the two inputs and the sign of the output; detection of overflow can then be formulated not in terms of the carry-out of the addition but by some simple rules. Assuming z is the value of the result $z = x + y$ after truncation has occurred, we have

$$\begin{array}{lll} x \text{ +ve and } y \text{ -ve} & & \rightarrow \text{ no overflow} \\ x \text{ -ve and } y \text{ +ve} & & \rightarrow \text{ no overflow} \\ x \text{ +ve and } y \text{ +ve and } z \text{ +ve} & & \rightarrow \text{ no overflow} \\ x \text{ +ve and } y \text{ +ve and } z \text{ -ve} & & \rightarrow \text{ overflow} \\ x \text{ -ve and } y \text{ -ve and } z \text{ +ve} & & \rightarrow \text{ overflow} \\ x \text{ -ve and } y \text{ -ve and } z \text{ -ve} & & \rightarrow \text{ no overflow} \end{array}$$

Overflow cannot occur if the input numbers have differing signs since the result *must* lie somewhere in the range of numbers representable. The overflow cases offer perhaps the most intuitive way to explain the rest: for example if we are adding two positive numbers together and get a negative result, something has clearly gone wrong. Given that x_{n-1} is the MSB of the value x and that this determines the sign of x in a twos-complement representation, we have that overflow occurs exactly when the expression

$$(x_{n-1} \wedge y_{n-1} \wedge \neg z_{n-1}) \vee (\neg x_{n-1} \wedge \neg y_{n-1} \wedge z_{n-1})$$

is true.

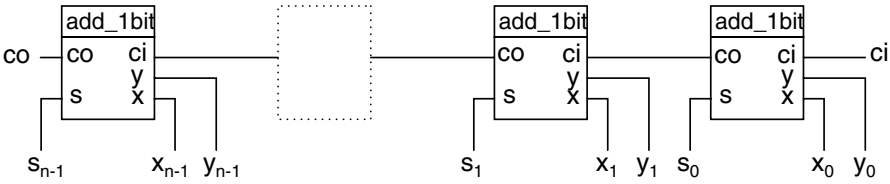


Figure 7.4 Design for an n-bit ripple-carry adder.

```
1  module add_nbit( co, ci, r, x, y );
2
3    parameter n = 8;
4
5    output wire         co;
6    input  wire         ci;
7
8    output wire [n-1:0] r;
9    input  wire [n-1:0] x;
10   input  wire [n-1:0] y;
11
12         wire [n-2:0] w0;
13
14   add_1bit t0( .co(w0[  0]), .s(r[  0]),
15                .ci(ci     ), .x(x[  0]), .y(y[  0]) );
16   add_1bit t1( .co(co     ), .s(r[n-1]),
17                .ci(w0[n-2]), .x(x[n-1]), .y(y[n-1]) );
18
19   genvar i;
20   generate
21     for( i = 1; i < n-1; i = i + 1 )
22     begin:gen_add
23       add_1bit t2( .co(w0[i  ]), .s(r[i]),
24                    .ci(w0[i-1]), .x(x[i]), .y(y[i]) );
25     end
26   endgenerate
27
28 endmodule
```

Listing 7.3 Implementation of an n-bit ripple-carry adder.

7.3.1.1 Ripple-Carry Adder

We already know how to construct half and full-adders from basic logic gates that are capable of adding two 1-bit values and a carry-in to give a sum and carry-out as a result. Our task in using this basic component to add larger n-bit values is fairly simple: we just need to chain together the carry-in and carry-out signals of n full-adders. This chaining technique was described previously in Chapter 2 where we built a 4-bit adder from 1-bit full-adder components.

Following this, Figure 7.4 shows the very similar, but general structure for an n-bit adder. This design is called a **ripple-carry adder** since the flow of carries through the adders mimics the flow of carries in our algorithm: the carries ripple or propagate from the i-th adder to the $(i+1)$-th adder. As such, the design can only

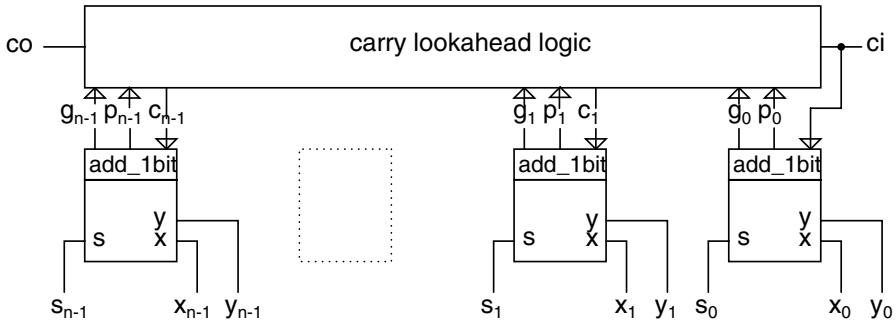


Figure 7.5 Design for an *n*-bit carry look-ahead adder.

calculate the *i*-th bit of the result once the adders to the right (i.e., those for the less-significant bits) have completed their results and propagated the carry. Implementing the design is very easy; Listing 7.3 shows a general-purpose *n*-bit adder module built assuming we have the standard `add_1bit` module available to us. To make the description more compact, we use a generate statement to instantiate the $n-2$ full-adders in the middle of the chain and only explicitly instantiate the two at either end which carry-in and carry-out `ci` and `co` respectively.

7.3.1.2 Carry Look-Ahead Adder

The goal of a **carry look-ahead adder** design is to reduce the propagation delay associated with the carry chain through a ripple-carry adder. The basic idea behind achieving this is to directly compute the carry-in for the *i*-th step rather than have it fed through from the $(i-1)$-th step.

We classify each step as either generating a carry-out or propagating a carry-in. Working in base-*b*, a step can generate a carry-out if $x_i + y_i \geq b$ while it can propagate a carry-in if $x_i + y_i = b - 1$. Consider an example with decimal, i.e., base-10, values:

$$\begin{array}{r} 1\ 4\ 5 \text{ input } x \\ 1\ 5\ 5 \text{ input } y \\ \hline 1\ 1\ 0 \text{ carry} \\ \hline 3\ 0\ 0 \text{ result } z = x + y \end{array}$$

The first step generates a carry-out because $5+5 \geq 10$ so the carry is passed into the next step. The second step propagates this carry; although it does not directly generate a carry-out since $4+5=9$, the carry-in from the previous step increments this such that the carry is again passed onto the next step. The final step neither generates nor propagates a carry since $1+1=2$ which, when incremented to accommodate the carry-in, is still less than the base-10.

In binary things are again a little more simple than in the general case. To determine if the i-th step of the sum $x+y$ generates or propagates a carry, which we denote by g_i and p_i respectively, we can use

$$\begin{aligned} g_i &= x_i \wedge y_i \\ p_i &= x_i \oplus y_i. \end{aligned}$$

That is, the step generates a carry-out if both digits are 1 and propagates a carry-in if either digit is 1 (but not both, otherwise it would generate a carry-out). Using these simple starting points, we can directly compute c_i, the carry-in for the i-th step. Given c_0 is the carry-in for the whole addition, we continue to find

$$\begin{aligned} c_1 &= g_0 \vee (p_0 \wedge c_0) \\ c_2 &= g_1 \vee (p_1 \wedge c_1) = g_1 \vee (p_1 \wedge g_0) \vee (p_1 \wedge p_0 \wedge c_0) \\ c_3 &= g_2 \vee (p_2 \wedge c_2) = g_2 \vee (p_2 \wedge g_1) \vee (p_2 \wedge p_1 \wedge g_0) \vee (p_2 \wedge p_1 \wedge p_0 \wedge c_0) \\ &\vdots \end{aligned}$$

Reading the first line, we have a carry-in for step one if step zero generates a carry-out or if step zero propagates a carry-in *and* there is a carry-in for step zero. Subsequent lines follow a similar reasoning, but are increasingly complex. However, using this approach we have significantly reduced the propagation delay associated with generating a result. Figure 7.5 demonstrates the general design of a carry look-ahead adder circuit.

As an aside, note that it is common to see g_i and p_i written as

$$\begin{aligned} g_i &= x_i \wedge y_i \\ p_i &= x_i \vee y_i. \end{aligned}$$

Of course, when used in the above this change does not alter the meaning: if $x_i = 1$ and $y_i = 1$, then $g_i = 1$ so it does not matter what the corresponding p_i is in this case. As such, use of an OR gate rather than an XOR is preferred because the former requires less transistors. In the ripple-carry adder case the circuit has a depth of $O(n)$ gates to generate a result, with carry look-ahead we reduce this depth to $O(\log n)$. The trade-off is that even though the circuit is less deep and hence generates a result faster, it generally uses many more gates: roughly $O(n^2)$ for a carry look-ahead adder versus $O(n)$ for a ripple-carry adder. As a compromise between these advantages and disadvantages, it can be attractive to chain together small carry look-ahead adders, say four 8-bit adders, in a ripple-carry configuration to form a larger, say a 32-bit, adder.

7.3.2 Subtraction

Subtraction is performed in a similar way to addition except we substitute carries from the i-th step to the $(i+1)$-th step into borrows from the $(i+1)$-th step by the

Input: The n-digit integers x and y in a base-b representation.
Output: The $n+1$-digit integer $z = x - y$ in a base-b representation.

```
1 SUB(x, y)
2     z ← 0
3     c ← 0
4     for i = 0 upto n − 1 step 1 do
5         t ← x_i − y_i + c
6         z_i ← t mod b
7         c ← ⌊t/b⌋
8     return z
9 end
```

Algorithm 7.2: An algorithm for multi-digit subtraction.

i-th step. In decimal we show this using the values $x = 218_{(10)}$ and $y = 123_{(10)}$ to make our life slightly easier, and execute Algorithm 7.2 to produce the trace

i	x_i	y_i	c	t	z_i	c
0	8	3	0	5	5	0
1	1	2	0	−1	9	−1
2	2	1	−1	0	0	0

and calculate $z = x - y = 95_{(10)}$. Written in the more conventional diagram form we get

$$\begin{array}{r} 2\ 1\ 8 \text{ input } x \\ 1\ 2\ 3 \text{ input } y \\ \hline -1\ 0\ 0 \text{ borrow} \\ \hline 9\ 5 \text{ result } z = x - y \\ \hline \end{array}$$

Translating this into the binary case results in exactly the same process; for our test values $x = 281_{(10)} = 11011010_{(2)}$ and $y = 123_{(10)} = 1111011_{(2)}$ we again compute $z = x + y = 95_{(10)} = 1011111_{(2)}$ as follows:

$$\begin{array}{rrrrrrrr} 1 & 1 & 0 & 1 & 1 & 0 & 1\ 0 & \text{input } x \\ & 1 & 1 & 1 & 1 & 0 & 1\ 1 & \text{input } y \\ \hline -1 & -1 & -1 & -1 & -1 & -1 & -1\ 0 & \text{borrow} \\ \hline & 1 & 0 & 1 & 1 & 1 & 1\ 1 & \text{result } z = x - y \\ \hline \end{array}$$

However, implementing separate addition and subtraction is somewhat pointless given we can deal with subtraction by using the identity $z = x - y = x + (-y)$. That is, we first compute the negation of y and then add this to x to get our result. At first glance it seems like we will still need an extra circuit to perform the negation, but in fact this can be achieved using the following convenient method to convert a

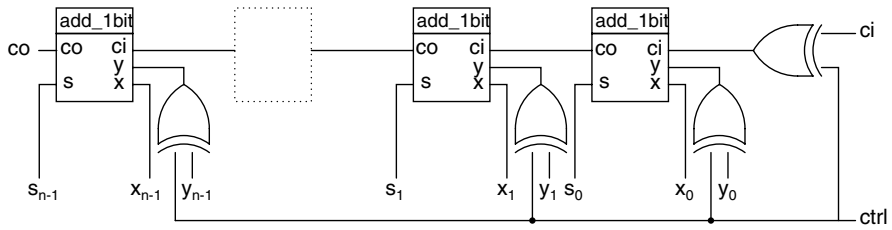


Figure 7.6 Design for an n-bit ripple-carry adder/subtractor.

number y into the twos-complement negation:

$$-y \equiv \neg y + 1.$$

So we simply invert each bit in y and then add one. However, we need to take care with regard to the carry-in since in subtraction this is essentially a borrow-from so we really want to compute $x + y + ci$ during addition and $x - y - ci$ during subtraction.

Inverting each bit of y is simple: we just have a control wire which determines if our adder should do an addition or a subtraction and XOR this control wire with each bit in y. As a result, when the control wire $ctrl = 1$ we have y_i inverted and when $ctrl = 0$ we pass y_i through unchanged. Adding one to the result is also simple: we just set the initial carry-in of our adder chain to 1. However, we need to take care to only do this when required; based on $ctrl$ and ci the four possible cases are

$$\begin{aligned}
&ctrl = 0 \text{ and } ci = 0 \rightarrow x + y + 0 && = x + y \\
&ctrl = 0 \text{ and } ci = 1 \rightarrow x + y + 1 && = x + y + 1 \\
&ctrl = 1 \text{ and } ci = 0 \rightarrow x + (\neg y + 1) - 0 && = x + (\neg y) + 1 \\
&ctrl = 1 \text{ and } ci = 1 \rightarrow x + (\neg y + 1) - 1 && = x + (\neg y)
\end{aligned}$$

Therefore, we actually set the initial carry-in of our adder chain to equal $ctrl \oplus ci$ so that it is 1 only where either we are doing an addition and there is a carry-in, or we are doing a subtraction and there is no borrow-from.

Figure 7.6 shows a design for a combined circuit that can perform addition and subtraction depending on a control signal. This design is implemented in Listing 7.4 and looks very similar to our initial adder implementation in that we again rely on the basic `add_1bit` module. This time, instead of feeding `y` into each adder as the second input, we use `w1` which is conditionally negated by using an XOR of each bit of `y` with the control value `ctrl`. This is accomplished using a replication operator to form an n-bit vector where each bit is equal to `ctrl`. Then, instead of feeding in `ci` as the carry-in to the first adder we use `ctrl` again so as to get an extra addition of 1 where negation of `y` is required. Actually, this is not quite true: to cope with the fact that we might still need to use the carry-in value in addition, we

```
module sub_nbit( ctrl, co, ci, r, x, y );

  parameter n = 8;

  input  wire         ctrl;

  output wire         co;
  input  wire         ci;

  output wire [n-1:0] r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         wire [n-2:0] w0;
         wire [n-1:0] w1 = y ^ {n{ctrl}};

  add_1bit t0( .co(w0[  0]  ), .s(r[  0]),
               .ci(ci ^ ctrl), .x(x[  0]), .y(w1[  0]) );
  add_1bit t1( .co(co       ), .s(r[n-1]),
               .ci(w0[n-2]  ), .x(x[n-1]), .y(w1[n-1]) );

  genvar i;
  generate
    for( i = 1; i < n-1; i = i + 1 )
    begin:gen_sub
      add_1bit t2( .co(w0[i  ]), .s(r[i]),
                   .ci(w0[i-1]), .x(x[i]), .y(w1[i]) );
    end
  endgenerate

endmodule
```

Listing 7.4 Implementation of an n-bit ripple-carry adder/subtractor.

actually XOR `ctrl` with `ci` to form the first carry-in. This allows us, for example, to set `ctrl` to 0 for an addition yet still set `ci` equal to 1.

7.4 Shift and Rotate

Formally, a left or right **shift** of an integer x represented in base-b by a distance y multiplies x by b^{+y} or b^{-y}. This basically has the effect of moving the digits left or right in the expansion. Consider an example value of $x = 123_{(10)}$. Working in decimal, shifting x left by setting $s = 2$ multiplies it by $10^2 = 100$ giving

$$12300_{(10)}$$

while shifting right by setting $y = -2$ multiplies it by $10^{-2} = 0.01$, i.e., it divides it by 100, to give

$$1.23_{(10)}.$$

This process works exactly the same in binary. So with $x = 1111011_{(2)}$, shifting x left by setting $y = 2$ multiplies x by $2^2 = 4$ to give

$$111101100_{(2)}$$

while shifting right by setting $y = -2$ multiplies x by $2^{-2} = 0.25$, i.e., it divides it by 4, to give

$$11110.11_{(2)}.$$

When representing numbers as n-bit binary vectors, there are some additional questions that need answering. For example, consider the vector $x = 01111011$ representing $1111011_{(2)}$ in 8 bits. Shifting left by setting $y = 2$ gives

$$111011??.$$

We have lost two digits from the left-hand, most-significant end because they cannot be accommodated in 8 bits. More importantly the right-hand, least-significant end of the vector needs two extra digits so it remains 8 bits long; these are marked with ?. The same is true when shifting in the other direction so that shifting right by setting $y = -2$ gives

$$??011110.$$

This time we have lost two digits from the right-hand, least-significant end and need two extra digits inserted into the left-hand, most-significant end.

How we fill the bits marked ? depends on the type of operation we require. Roughly, there are three important operations: **logical shifts**, **arithmetic shifts**, and **rotations**. These are best explained by example, starting with operations in the left direction:

- If we are performing a logical left-shift, we fill the least-significant bits of the result with 0. Thus, a logical left-shift of the vector $x = 01111011$ by $y = 2$ results in 11101100.
- If we are performing a left-rotate, we fill the least-significant bits of the result with the most-significant bits expelled from the most-significant end. Thus, a logical left-shift of the vector $x = 01111011$ by $y = 2$ results in 11101101.

Similar behaviour can be applied to operations in the right direction:

- If we are performing a logical right-shift, we fill the most-significant bits of the result with 0. Thus, a logical right-shift of the vector $x = 01111011$ by $y = -2$ results in 00011110.
- If we are performing an arithmetic right-shift, we fill the most-significant bits of the result with the most-significant or sign bit. Thus, an arithmetic right-shift of the vector $x = 01111011$ by $y = -2$ results in 00011110 while using the value $x = 11111011$ results in 11111110. This preserves the sign so we get the result we would expect if viewing right-shift as a form of division.
- If we are performing a right-rotate, we fill the most-significant bits of the result with the least-significant bits expelled from the least significant end. Thus, a logical right-shift of the vector $x = 01111011$ by $y = 2$ results in 11011110.

The idea of differentiating between logical and arithmetic shifts is that arithmetic shifts preserve the sign of the value involved while logical shifts do not. This can be

important depending on the context in which the operation is used: if we really want to view right-shifting as a division by 2^y, we need an arithmetic shift otherwise the sign of the result will be wrong.

Once we know what the various operations should actually do, we can start to think about ways of implementing them in hardware. There are two main categories of implementation which are described in detail below. To make things easier for ourselves, we will only consider shifting values by unsigned distances. The assumption is that the device controlling our shifter will translate, for example, a logical left-shift by a negative distance into a logical right-shift by a positive distance of the same magnitude if required. Thus the assumption is that for an n-bit value x we always have the distance $0 \leq y < \log_2(n)$.

Finally, we will assume that the style of shift performed by our hardware is dictated by some input which can take one of the following constant values:

```
`define SHF_R_LOGIC  3'b000
`define SHF_R_ARITH  3'b001
`define SHF_R_ROTATE 3'b010
`define SHF_L_LOGIC  3'b011
`define SHF_L_ARITH  3'b100
`define SHF_L_ROTATE 3'b101
```

with `` `SHF_L_ARITH `` being an unused placeholder.

7.4.1 Bit-Serial Shifter

The most simple way to realise the required behaviour is to use a sequential or bit-serial approach. The basic idea is to build a hardware device capable of shifting values by a distance of one bit and then iterating the use of this hardware using y steps to shift a distance of y bits. The advantage of this approach is that it uses the least logic; we only require hardware to shift by distances of one bit coupled to some control hardware. The drawback is that it is slow to produce a result; we must make y sequential steps to get the required result.

Listing 7.5 details the implementation of a simple **bit-serial shifter**. The module uses the input `oper` to decide which type of operation is required; one can perform left or right logical or arithmetic shifts and rotations. In order to realise the iterated behaviour, we maintain a counter in register `r1` that tracks how many iterations we have performed so far. This value is incremented by `add_nbit` instance `t0` and compared with the desired number of iterations by `equ_nbit` instance `t1`.

Execution of a shift operation is initialised when a positive edge is detected on the reset input named `rst`. When this event occurs we reset the iteration counter to zero and the temporary value `r0`, which stores the value being shifted, to the initial input `x`. On positive edges of the `clk` signal the shifter state is updated; that is, one iteration is executed. We start by updating our temporary shifted value `r0` by shifting it by a distance of one bit in a style determined by `oper`. We then check if enough iterations have been performed by examining `w1`, the result of comparing the

```
module shf_nbit( clk, rst, oper, r, x, y );

  parameter n = 8;
  parameter m = 3;

  input  wire         clk;
  input  wire         rst;

  input  wire [  2:0] oper;

  output reg  [n-1:0] r;
  input  wire [n-1:0] x;
  input  wire [n-1:0] y;

         reg  [n-1:0] r0;
         reg  [m-1:0] r1;
         reg          r2;

         wire [m-1:0] w0;
         wire         w1;

  add_nbit t0(.ci(1'b0),.r(w0),.x(r1),.y(1       ) );
  equ_nbit t1(          .r(w1),.x(w0),.y(y[m-1:0]) );

  defparam t0.n = m;
  defparam t1.n = m;

  always @ ( posedge rst )
  begin
    r0 = x;
    r1 = 0;
    r2 = 0;
  end

  always @ ( posedge clk )
  begin
    case( oper )
      `SHF_R_LOGIC  : r0 = { 1'b0,    r0[n-1:1] };
      `SHF_R_ARITH  : r0 = { r0[n-1], r0[n-1:1] };
      `SHF_R_ROTATE : r0 = { r0[  0], r0[n-1:1] };
      `SHF_L_LOGIC  : r0 = { r0[n-2:0], 1'b0    };
      `SHF_L_ROTATE : r0 = { r0[n-2:0], r0[n-1] };
    endcase

    if( !r2 )
    begin
      r1 = w0;

      if( w1 )
      begin
        r  = r0;

        r2 =  1;
      end
    end
  end

endmodule
```

Listing 7.5 Implementation of an *n*-bit bit-serial shifter.

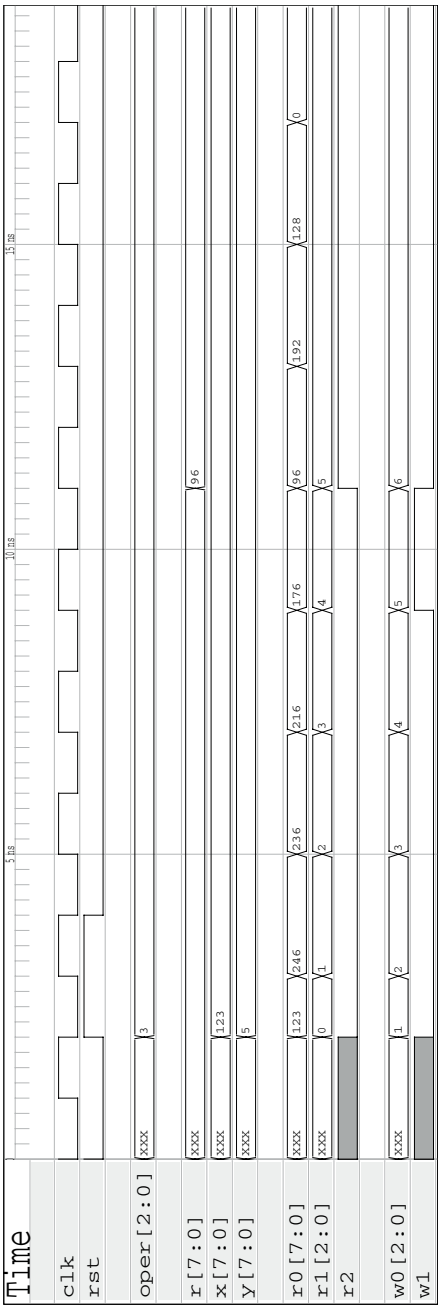


Figure 7.7 Behaviour of an n-bit bit-serial shifter.

iteration counter with the desired number of iterations. If we have done enough, we set the shifter output `r` to equal our temporary value `r0` to complete the operation. Otherwise we update the iteration counter by incrementing it and wait for `clk` to trigger another iteration.

Figure 7.7 demonstrates the behaviour of this design for the inputs $x = 123_{(10)} = 1111011_{(2)}$ and $y = 5_{(10)} = 101_{(2)}$ when asked to perform a left logical shift. The clock has a period of 2ns and the shifter is reset at time $t = 2$ns by the positive edge of `rst`. Then, on each positive edge of the clock one can see the loop counter, accumulator and so on being updated. By time $t = 12$ns the result is set as $r = 96_{(10)} = 1100000_{(2)}$ which is what we expect; the left-most five bits of x have been shifted off and zeros inserted as padding on the right-hand side.

7.4.2 Logarithmic Shifter

In reality, it is often desirable to generate results quickly rather than to minimise the amount of logic used. As such, one can consider alternative designs that by using more logic can produce a result in just one step rather than y steps. This approach demands that we make a combinatorial circuit rather than a clocked circuit; the speed one can generate results is then dominated by the critical path of the circuit rather than the clock speed. We consider a conceptually simple approach which is termed a **logarithmic shifter**. Essentially this design is a series of multiplexers which shift the input by differing constant amounts to form the result.

The key idea is that each bit of the shift distance y determines a shift by a constant amount 2^i. So for example, if $y_1 = 1$ then we need to shift by a distance of two at some point. The combination of the different constant shifts results in the overall result so if $y = 3_{(10)} = 11_{(2)}$, we have $y_0 = 1$ and $y_1 = 1$ and so need to shift by a distance of one then a distance of two. Since the different constant shifts are independent, we can do them separately and have one unit shifting by 1 then feed this result to a unit shifting by two. Both units are easy to construct since the shift distances are now constant. The end result is a structure shown in Figure 7.8. The design is combinatorial and produces the end result in one clock cycle; the obvious drawback is the amount of logic used and longer associated critical path.

Implementing a general n-bit logarithmic shifter in Verilog is quite hard due to the constraints on use of generate statements; one could easily write a C program to generate the required code however. To provide a concrete example, we present the implementation of a specific 8-bit shifter. Hopefully it is clear that similar techniques can easily be applied to larger designs. Listing 7.6 details the implementation which consists of three similar stages. Each stage consists of two multiplexers: a `mul8_8bit` instance which shifts the input by a constant distance using a style dictated by `oper`, and a `mul2_8bit` instance which decides between the shifted and unshifted input depending on the associated bit of input `y`. So for example multiplexer instance `t0` shifts the input `w0`, an alias for `x`, by a distance of one bit. Multiplexer instance `t1` then decides whether to feed the original value `w0` or the

```
module shf_8bit( input  wire [2:0] oper,

                 output wire [7:0] r,
                 input  wire [7:0] x,
                 input  wire [7:0] y );

  wire [7:0] w0;
  wire [7:0] w1;
  wire [7:0] w2;
  wire [7:0] w3;
  wire [7:0] w4;
  wire [7:0] w5;
  wire [7:0] w6;

  assign w0 =  x;
  assign r  = w6;

  mux8_8bit t0( .r(w1),

                .i0({ 1'b0,          w0[7:1] }),
                .i1({ {1{w0[7:7]}}, w0[7:1] }),
                .i2({ w0[0:0],       w0[7:1] }),
                .i3({ w0[6:0],       1'b0    }),
                .i5({ w0[6:0],       w0[7:7] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t1( .r(w2), .i0(w0), .i1(w1), .s0(y[0]) );

  mux8_8bit t2( .r(w3),

                .i0({ 2'b0,          w2[7:2] }),
                .i1({ {2{w2[7:7]}}, w2[7:2] }),
                .i2({ w2[1:0],       w2[7:2] }),
                .i3({ w2[5:0],       2'b0    }),
                .i5({ w2[5:0],       w2[7:6] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t3( .r(w4), .i0(w2), .i1(w3), .s0(y[1]) );

  mux8_8bit t4( .r(w5),

                .i0({ 4'b0,          w4[7:4] }),
                .i1({ {4{w4[7:7]}}, w4[7:4] }),
                .i2({ w4[3:0],       w4[7:4] }),
                .i3({ w4[3:0],       4'b0    }),
                .i5({ w4[3:0],       w4[7:4] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t5( .r(w6), .i0(w4), .i1(w5), .s0(y[2]) );

endmodule
```

Listing 7.6 Implementation of an 8-bit logarithmic shifter.

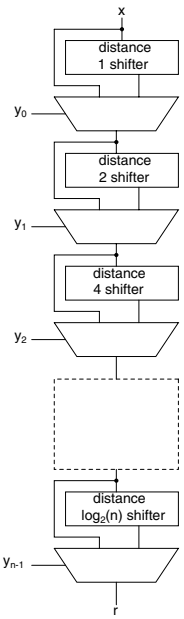


Figure 7.8 General design of a basic logarithmic shifter architecture.

shifter value `w1` through to the result `w2` depending on the value of `y[0]`. The next stage, comprising the multiplexer instances `t2` and `t3`, performs a similar operation on `w2` to produce `w4` but for a shift distance of two bits; `t4` and `t5` then shift `w4` by a distance of four bits to produce `w6`. The end result `w6` is the correctly shifted value and is assigned to the shifter output `r`.

7.5 Multiplication

Like addition, most people learn the method for multi-digit multiplication in school. When calculating $z = x \cdot y$ we term x the **multiplicand** and y the **multiplier**; since multiplication is associative we can switch these roles if we want. To make things easier for ourselves, we will only consider multiplication of unsigned numbers. Using some extra logic one can easily deal with signed input by forcing the inputs to be unsigned and then fixing up the sign of the result.

In a similar way to addition, we can describe multiplication using Algorithm 7.3 and produce a trace of execution for our example inputs $x = 123_{(10)}$ and $y = 218_{(10)}$

Input: The n-digit integers x and y in a base-b representation.
Output: The $n+1$-digit integer $z = x \cdot y$ in a base-b representation.

```
1  MUL(x, y)
2      z ← 0
3      for i = 0 upto n − 1 step 1 do
4          c ← 0
5          for j = 0 upto n − 1 step 1 do
6              t ← x_i · y_j + z_{i+j} + c
7              z_{i+j} ← t mod b
8              c ← ⌊t/b⌋
9          z_{i+w} ← c
10     return z
11 end
```

Algorithm 7.3: An algorithm for multi-digit multiplication.

i	j	x_i	y_j	z_{i+j}	c	t	z_{i+j}	c
0	0	3	8	0	0	24	4	2
0	1	3	1	0	2	5	5	0
0	2	3	2	0	0	6	6	0
1	0	2	8	5	0	21	1	2
1	1	2	1	6	2	10	0	1
1	2	2	2	0	1	5	5	0
2	0	1	8	0	0	8	8	0
2	1	1	1	5	0	6	6	0
2	2	1	2	0	0	2	2	0

Alternatively, we can describe the operation diagrammatically as

```
    1 2 3  input x
    2 1 8  input y
  ---------
    9 8 4  partial product for +x · 8
  1 2 3    partial product for +x · 10
2 4 6      partial product for +x · 200
---------
2 6 8 1 4  result z = x · y
```

At the i-th step we produce a partial product equal to $(y_i \cdot x) \cdot b^i$. That is, we take the i-th digit of y, multiply it by x and then shift it along i places into the right column to form the partial product. The partial products are then added together to produce the result. As with all our algorithms we can work in binary just as easily as decimal and thus for $x = 123_{(10)} = 1111011_{(2)}$ and $y = 211_{(10)} = 11011010_{(2)}$ we have

Input: The n-bit integers x and y in a base-2 representation.
Output: The $2n$-bit integer $z = x \cdot y$ in a base-2 representation.

```
1 MUL-BIT-SERIAL(x, y)
2     t ← 0
3     for i = n − 1 downto 0 step −1 do
4         t ← 2 · t
5         if y_i = 1 then
6             t ← t + x
7     return t
8 end
```

Algorithm 7.4: A bit-serial algorithm for multiplication of binary numbers.

```
                  0 1 1 1 1 0 1 1   input x
                  1 1 0 1 1 0 1 0   input y
                ---------------------------------------------
                  0 0 0 0 0 0 0 0   partial product for +x · 0
                0 1 1 1 1 0 1 1     partial product for +x · 10
              0 0 0 0 0 0 0 0       partial product for +x · 000
            0 1 1 1 1 0 1 1         partial product for +x · 1000
          0 1 1 1 1 0 1 1           partial product for +x · 10000
        0 0 0 0 0 0 0 0             partial product for +x · 000000
      0 1 1 1 1 0 1 1               partial product for +x · 1000000
    0 1 1 1 1 0 1 1                 partial product for +x · 10000000
  ---------------------------------------------
  1 1 0 1 0 0 0 1 0 1 1 1 1 1 1 0   result z = x · y
```

One minor issue that presents itself immediately from this description is that given two n-bit inputs, any multiplier design we select will generate a $2n$-bit result. This differs from addition and logical operations which both generate n-bit results if one ignores the carry-out. This is not a major problem: we just need an extra output from the ALU so we can return the high and low halves of the result to the processor core from the multiplier.

Either way, since multiplication is one of the most costly operations it is worthwhile considering the best way to realise the functionality of Algorithm 7.3 in hardware. There are a large number of different implementation options but most multipliers fall roughly into one of three categories which are described in detail below.

7.5.1 Bit-Serial Multiplier

At face value, the description of multiplication in binary above seems much more complicated than the decimal case. However, it is actually made far easier because at the i-th step, the i-th bit of the multiplier can only be zero or one: we simply either

```
module mul_nbit( clk, rst, r, x, y );

  parameter n = 8;
  parameter m = 3;

  input  wire           clk;
  input  wire           rst;

  output reg  [2*n-1:0] r;
  input  wire [  n-1:0] x;
  input  wire [  n-1:0] y;

         reg  [2*n-1:0] r0;
         reg  [  n-1:0] r1;
         reg  [  m-1:0] r2;
         reg            r3;

         wire [  m-1:0] w0;
         wire           w1;
         wire [2*n-1:0] w2 = { r0[2*n-2:0], 1'b0 };
         wire [2*n-1:0] w3;

  add_nbit t0(.ci(1'b0),.r(w0),.x(r2),.y(               1));
  equ_nbit t1(          .r(w1),.x(r2),.y(            n-1));
  add_nbit t2(.ci(1'b0),.r(w3),.x(w2),.y({ {n{1'b0}}, x }));

  defparam t0.n =   m;
  defparam t1.n =   m;
  defparam t2.n = 2*n;

  always @ ( posedge rst )
  begin
    r0 = 0;
    r1 = y;
    r2 = 0;
    r3 = 0;
  end

  always @ ( posedge clk )
  begin
    if( r1[n-1] )
      r0 = w3;
    else
      r0 = w2;

    r1 = { r1[n-2:0], 1'b0 };

    if( !r3 )
    begin
      r2 = w0;

      if( w1 )
      begin
        r  = r0;
        r3 =  1;
      end
    end
  end

endmodule
```

Listing 7.7 Implementation of an n-bit bit-serial multiplier.

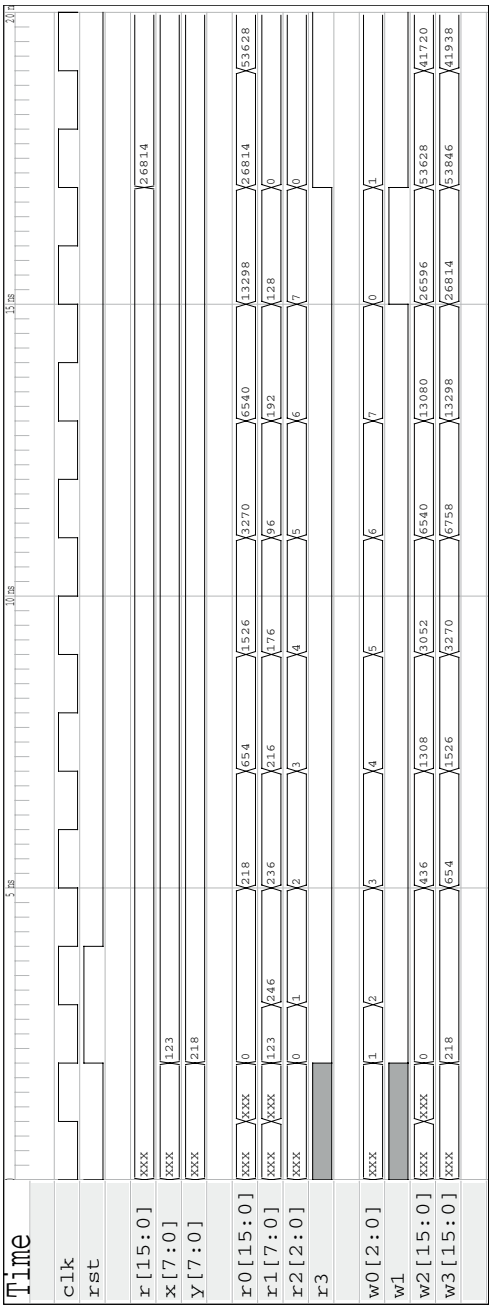


Figure 7.9 Behaviour of an n-bit bit-serial multiplier.

add the partial product into the accumulated total or not. Combined with the fact that the partial product at the i-th step is simply the multiplicand shifted left i digits, we can formulate the bit-serial method for multiplication in Algorithm 7.4. To see why this algorithm works, we need to think about what multiplication "means". One way to look at it is that multiplication is just repeated addition. For example

$$x \cdot y = \underbrace{x + x + \cdots + x + x}_{\text{total of } y \text{ terms}}$$

so if we select $y = 6_{(10)}$, then we obviously have

$$x \cdot 6 = x + x + x + x + x + x.$$

But there is a clear drawback to this in the sense that the bigger y is, the longer that list of terms is and hence the more additions we are going to need to do if we actually want to compute $x \cdot y$. Another way of looking at it is that multiplication adds another "weight" into the summation above. It might look odd, but imagine we wrote y out in binary; then we could write $x \cdot y$ as

$$x \cdot y = x \cdot \sum_{i=0}^{n-1} y_i \cdot 2^i.$$

Again selecting $y = 6_{(10)} = 110_{(2)}$ we find that we still get the result we would expect to:

$$\begin{aligned} x \cdot y &= x \cdot y_2 \cdot 2^2 + x \cdot y_1 \cdot 2^2 + x \cdot y_0 \cdot 2^0 \\ &= x \cdot 1 \cdot 2^2 + x \cdot 1 \cdot 2^2 + x \cdot 0 \cdot 2^0 \\ &= x \cdot 4 + x \cdot 2 + x \cdot 0 \\ &= x \cdot 6 \quad . \end{aligned}$$

The problem is, this still looks unpleasant to compute. For example, we keep having to compute powers of two to "weight" the terms. Fortunately, British mathematician William Horner worked out a scheme to do this more neatly. Basically we just bracket the thing we started with in such a way that instead of having to compute the powers of two independently, we sort of accumulate them; this is best shown by example:

$$\begin{aligned} x \cdot y &= ((x \cdot y_2) \cdot 2 + x \cdot y_1) \cdot 2 + x \cdot y_0 \\ &= ((x \cdot 1) \cdot 2 + x \cdot 1) \cdot 2 + x \cdot 0 \\ &= (x \cdot 2 + x \cdot 1) \cdot 2 + x \cdot 0 \\ &= x \cdot 4 + x \cdot 2 + x \cdot 0 \\ &= x \cdot 6 \quad . \end{aligned}$$

Working inside out (i.e., from the inner-most term in the nest of brackets to the outer-most) the computation no longer looks so unpleasant, plus we can see the relationship to Algorithm 7.4. That is, at the i-th step we take the accumulated value and double it then add x if $y_i = 1$.

As one can imagine, this approach uses the least area of our three choices since it requires only an adder to add partial products into the accumulator, a shifter to

ensure the partial products are added into the right place, and some control logic to execute the loop. However, it also yields the worst performance in that it takes us n steps to multiply two n-bit values. Listing 7.7 describes an implementation of Algorithm 7.4 but requires some explanation. The multiplier is reset to an initial state on positive edges of the `rst` signal. On positive edges of the `clk` signal the multiplier state is updated; one can think of this as executing one iteration of the loop on each tick of the clock. To control the loop we use register `r2` as the loop counter, incrementing it via `add_nbit` instance `t0` and comparing it with the loop bound via `equ_nbit` instance. The register `r0` is used to store the accumulator from Algorithm 7.4. The wire `w2` represents `r0` shifted left by one bit, i.e., multiplied by two. On each iteration of the loop we update `r0` with either `w3`, the result of adding `x` to `w2` via the `add_nbit` instance `t2`, or `w2` itself. That is, on each iteration of the loop we either double `r0` or double it and add `x` depending on the related bit in `y`. To make life easier, we store `y` in register `r1` and shift the value left one bit each time the loop is iterated; this means we always inspect bit $n-1$ of `r1` rather than bit i of `y`. Finally, when we have done enough iterations the accumulator `r0` is assigned to the multiplier result `r` to complete the operation.

The behaviour of this design when fed our example inputs $x = 123_{(10)}$ and $y = 218_{(10)}$ is shown in Figure 7.9. The clock has a period of 2ns and the multiplier is reset at time $t = 2$ns by the positive edge of `rst`. Then, on each positive edge of the clock one can see the loop counter, accumulator and so on being updated. At time $t = 16$ns the final loop iteration completes and we output the accumulated result on `r`, the multiplier is then ready for any subsequent operations required of it.

7.5.2 Tree Multiplier

A parallel multiplier attempts to achieve high-performance by using the largest possible area of our three options. An example of this type is a **tree multiplier**; the basic idea is to generate all the partial products in parallel and then implement a large combinatorial circuit to add them together. This means that although we use much more logic than a bit-serial multiplier, we generate a result in one step and in a time bounded by propagation delay through the adders.

In implementing such a multiplier, instead of using a linear chain of n adders to accumulate the n partial products we reduce the associated propagation delay by using a tree. The result is the delay before a result is produced is $O(\log n)$ rather than $O(n)$. The general outline for such a design is shown in Figure 7.10. One can see how the n partial products would be accumulated by a tree of depth $\log_2 n$. The i-th partial product itself is constructed by ANDing input x with a vector replicated from the i-th bit of input y; the result is x if we want to add this partial product to the final result and zero otherwise.

Like the logarithmic shifter, describing a general n-bit parallel multiplier in Verilog is quite difficult; we again implement an example 8-bit design in Listing 7.8. The first block of code defines wires `w0` to `w7` and uses them to represent the eight

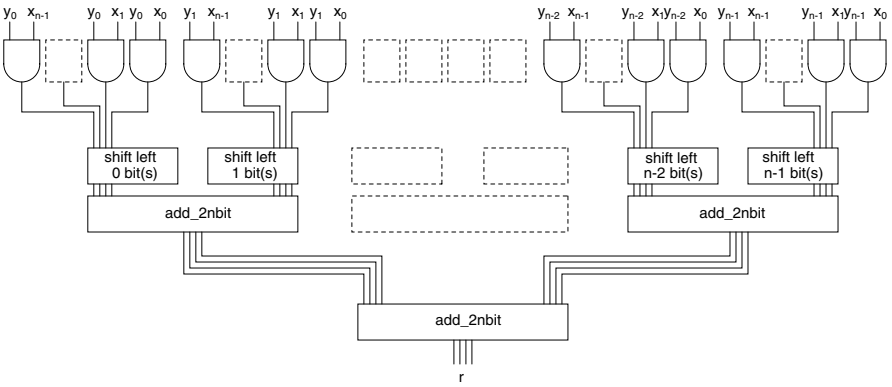


Figure 7.10 Design for a basic parallel multiplier architecture.

partial products. The i-th partial product is shifted left by i bits to place it in the right position for accumulation. The partial products are then accumulated by the second block of code which defines `t0` to `t6`, a tree of `add_nbit` instances which outputs the final result in `r`. Note that in this simple example we have padded all our partial products to 16 bits and parametrised our adder tree accordingly. In reality this is somewhat wasteful since, for example, in the addition of `w0` and `w1` we know some bits of either input are zero: we know that bit eight upwards of `w0` and bit zero of `w1` are all zero so this is a "special" addition in a sense. The bottom line is that by eliminating unnecessary operations that can be identified at design-time, we can reduce the overall cost in terms of logic.

7.5.3 Digit-Serial Multiplier

One way to view the previous designs is that a bit-serial approach deals with the multiplier one bit at a time while the parallel approach deals with all n multiplier bits at once. The natural compromise lies somewhere between these extremes and deals with say d bits in parallel with $1 < d < n$. We term such a design a digit-serial multiplier; d is said to be the digit size and is normally selected to divide n exactly to simplify implementation.

Algorithm 7.5 presents a rough outline of the algorithm where we use **par** to denote parallel execution of a sequence of statements. The basic idea is that we now step through the outer loop in steps of d instead of steps of 1 as in the original bit-serial version; in each iteration of the loop we accumulate d partial products in parallel. Thus for around a d-fold increase in size, we produce a d-fold increase in performance. This is a nice trade-off since we can scale d to match our size budget or our performance goals.

```
module mul_8bit( output wire [15:0] r,
                 input  wire [ 7:0] x,
                 input  wire [ 7:0] y );

  wire [15:0] w0 = {8'b0, {8{y[0]}} & x}     ;
  wire [15:0] w1 = {8'b0, {8{y[1]}} & x} << 1;
  wire [15:0] w2 = {8'b0, {8{y[2]}} & x} << 2;
  wire [15:0] w3 = {8'b0, {8{y[3]}} & x} << 3;
  wire [15:0] w4 = {8'b0, {8{y[4]}} & x} << 4;
  wire [15:0] w5 = {8'b0, {8{y[5]}} & x} << 5;
  wire [15:0] w6 = {8'b0, {8{y[6]}} & x} << 6;
  wire [15:0] w7 = {8'b0, {8{y[7]}} & x} << 7;

  wire [15:0] s0;
  wire [15:0] s1;
  wire [15:0] s2;
  wire [15:0] s3;
  wire [15:0] s4;
  wire [15:0] s5;

  add_nbit t0(.ci(1'b0),.r(s0),.x(w0),.y(w1));
  add_nbit t1(.ci(1'b0),.r(s1),.x(w2),.y(w3));
  add_nbit t2(.ci(1'b0),.r(s2),.x(w4),.y(w5));
  add_nbit t3(.ci(1'b0),.r(s3),.x(w6),.y(w7));

  add_nbit t4(.ci(1'b0),.r(s4),.x(s0),.y(s1));
  add_nbit t5(.ci(1'b0),.r(s5),.x(s2),.y(s3));

  add_nbit t6(.ci(1'b0),.r(r ),.x(s4),.y(s5));

  defparam t0.n = 16;
  defparam t1.n = 16;
  defparam t2.n = 16;
  defparam t3.n = 16;

  defparam t4.n = 16;
  defparam t5.n = 16;

  defparam t6.n = 16;

endmodule
```

Listing 7.8 Implementation of an 8-bit parallel multiplier.

7.5.4 Early Termination

To reduce the number of iterations taken by the basic bit-serial multiplier, one can consider the concept of **early termination** or **early exit**. We demonstrate this concept as a continuation of our left-to-right bit-serial multiplier from above. The basic idea is to check at some iteration i if the remaining bits of y are zero, i.e., $y_{i\ldots 0} = 0$. In this case we would simply iterate through the remaining bits always performing the operation $t \leftarrow 2 \cdot t$ but never performing the operation $t \leftarrow t + x$. So instead, we can exit the loop and scale t in one go, returning $t \cdot 2^i$ and thereby replicating the effect of i iterations of the operation $t \leftarrow t + t$. This is inexpensive because the scaling can be achieved simply by shifting t by a known distance. The benefit is that we do not do all those extra iterations, so potentially we can save some time.

Input: The n-bit integers x and y in a base-2 representation, a digit-size d.
Output: The $2n$-bit integer $z = x \cdot y$ in a base-2 representation.

```
1 MUL-DIGIT-SERIAL(x, y)
2     t ← 0
3     for i = n − 1 downto 0 step −d do
4         t ← 2^d · t
5         par j = 0 upto d − 1 step 1
6             if y_{i+j} = 1 then
7                 t ← t + x · 2^j
8     return t
9 end
```

Algorithm 7.5: A digit-serial algorithm for multiplication of binary numbers.

Input: The n-bit integers x and y in a base-2 representation, an integer block size $b|n$.
Output: The $2n$-bit integer $z = x \cdot y$ in a base-2 representation.

```
1 MUL-BIT-SERIAL-ET(x, y)
2     for i = n/b downto 1 step −1 do
3         if y_{b·i−1...0} = 0 then
4             return 2^{b·i} · t
5         else
6             for j = 0 upto b − 1 step 1 do
7                 t ← 2 · t
8                 if y_{b·i−1−j} = 1 then
9                     t ← t + x
10    return t
11 end
```

Algorithm 7.6: A bit-serial algorithm for multiplication of binary numbers with early termination.

Algorithm 7.6 roughly models what is going on. The idea is to perform the multiplication in n/b steps; in the case of $n = 32$ we might set $b = 8$ for example to perform four steps, dealing with eight bits of y in each step. The first thing we do in each step is check whether the remaining bits of y are all zero: if they are, the scaled t is returned straight away, otherwise we continue more or less as in the case of bit-serial multiplication.

Some examples might illustrate better what is going on. Imagine we use the values $n = 32$ and $b = 8$, and consider the inputs $x = 5_{(10)}$ and $y = 6_{(10)} = 00000006_{(16)}$; by writing y in hexadecimal, it is easy to see that the bottom 8-bit chunk is non-zero.

Therefore, when the algorithm checks the remaining bits of y at each of the four sets, there are still some which are non-zero so it behaves the same as the original bit-serial algorithm:

i	j	$8 \cdot i - 1 - j$	$y_{8 \cdot i - 1 - j}$	t	t
4	0	31	0	0	$0+0 = 0$
4	1	30	0	0	$0+0 = 0$
4	2	29	0	0	$0+0 = 0$
4	3	28	0	0	$0+0 = 0$
⋮	⋮	⋮	⋮	⋮	⋮
1	4	3	0	0	$0+0 = 0$
1	5	2	1	0	$0+0+5 = 5$
1	6	1	1	5	$5+5+5 = 15$
1	7	0	0	15	$15+15 = 30$

Now consider the inputs $x = 5_{(10)}$ and $y = 1536_{(10)} = 0000600_{(16)}$:

i	j	$8 \cdot i - 1 - j$	$y_{8 \cdot i - 1 - j}$	t	t
4	0	31	0	0	$0+0 = 0$
4	1	30	0	0	$0+0 = 0$
4	2	29	0	0	$0+0 = 0$
4	3	28	0	0	$0+0 = 0$
⋮	⋮	⋮	⋮	⋮	⋮
2	4	11	0	0	$0+0 = 0$
2	5	10	1	0	$0+0+5 = 5$
2	6	9	1	5	$5+5+5 = 15$
2	7	8	0	15	$15+15 = 30$

At this point, i.e., the final step, the algorithm notices the least-significant 8-bit chunk of y is zero so returns the value $30 \cdot 2^8$ (which is $7680_{(10)}$ as expected) without performing the loop iterations associated with the final step.

Some ARM processors use early termination but combine this with some extra techniques and use a right-to-left approach rather than left-to-right as we have done. The reasoning for this is simple: in a normal workload, most multipliers will be small and hence their more-significant bits are more likely to be zero than their less-significant bits. For small multipliers (e.g., loop counters, array indices and so on) the left-to-right approach will yield a higher chance of early termination, and hence greater reduction in the number of iterations.

7.5.5 Wallace and Dadda Trees

Tree multipliers offer a good trade-off in favour of performance versus size. However, a drawback of such designs is the long critical path which runs through the tree

	Stage 1	Stage 2		Stage 3	
	Output	Action	Output	Action	Output
Weight-1	1	*PT*	1	*PT*	1
Weight-2	2	*HA*	1	*PT*	1
Weight-4	3	*FA*	2	*HA*	1
Weight-8	4	*FA*	3	*FA*	2
Weight-16	3	*FA*	2	*HA*	2
Weight-32	2	*HA*	2	*HA*	2
Weight-64	1	*PT*	2	*HA*	2
Weight-128	0		0		1

(a) Actions in constructing Wallace multiplier for 4-bit inputs.

	Stage 1	Stage 2		Stage 3	
	Output	Action	Output	Action	Output
Weight-1	1	*PT*	1	*PT*	1
Weight-2	2	*PT*	2	*PT*	2
Weight-4	3	*FA*	1	*PT*	1
Weight-8	4	*FA*	3	*FA*	1
Weight-16	3	*FA*	2	*HA*	2
Weight-32	2	*PT*	3	*FA*	2
Weight-64	1	*PT*	1	*PT*	2
Weight-128	0		0		0

(b) Actions in constructing Dadda multiplier for 4-bit inputs.

Table 7.1 A tabular description of the stages in Wallace and Dadda multipliers for 4-bit inputs.

of adders; if we use ripple-carry adders to build an n-bit tree adder, the number of 1-bit full-adders the critical path passes through is essentially $O(n\log_2 n)$.

A **carry-save** adder takes a different approach to ripple-carry and carry look-ahead adders: instead of computing an addition per-se, it compresses three n-bit inputs x, y and z into two n-bit outputs. That is, it computes s and c, which we call the partial sum and shifted carry, where

$$\begin{aligned} s_i &= x_i \oplus y_i \oplus z_i \\ c_i &= (x_i \wedge y_i) \vee (x_i \wedge z_i) \vee (y_i \wedge z_i). \end{aligned}$$

Sometimes this structure is called a $3:2$ **compressor**. To compute the real $(n+2)$-bit sum of x, y and z one needs to combine s and c in a further addition step by computing $2c+s$ using a standard (ripple-carry for example) adder. The idea is that use of a carry-save adder followed by a ripple-carry adder is faster than two sequential uses of ripple-carry adders.

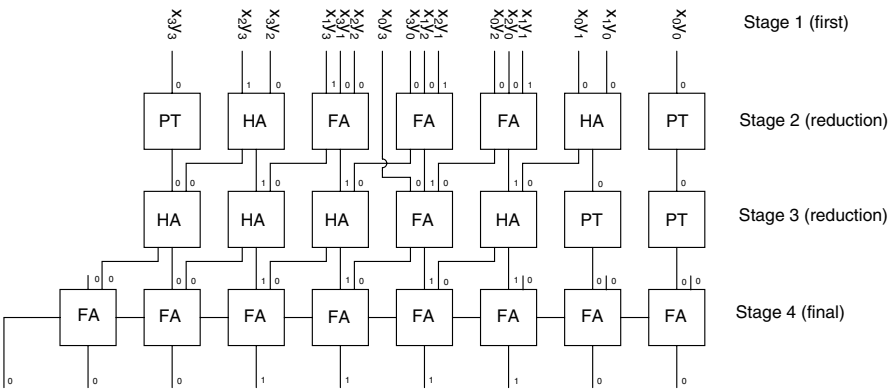


Figure 7.11 An example 4-bit Wallace multiplier.

Wallace multiplier [63] and **Dadda multiplier** [19] designs capitalise on the concept of carry-save adders. Both are tree-like multiplier designs in the sense that they generate and sum partial products in parallel, but improve the critical path through the multiplier. For n-bit inputs, both are constructed from a number of stages: an initial stage to generate the partial products; one or more reduction stages which act to reduce the partial products into two $2n$-bit values; and a final stage which adds the $2n$-bit values to produce the result. With a Wallace multiplier the basic approach to multiplying x and y is as follows:

1. In the first stage, multiply together (i.e., AND together) each x_i with each y_j to produce a total of n^2 intermediate wires. Each wire is said to have a weight, for example $x_0 \cdot y_0$ has weight 1 as $2^0 \cdot 2^0 = 1$, $x_1 \cdot y_2$ has weight 8 as $2^1 \cdot 2^2 = 8$.
2. Reduce the number of intermediate wires using additional stages composed of full-adders and half-adders:
   - Combine any three wires with same weight using a full-adder; result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
   - Combine any two wires with same weight using a half-adder; result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
   - If there is only one wire with a given weight, just pass it through to the next stage.
3. In the final stage, all weights have just one or two wires in them: combine the wires into two $2n$-bit values and add them with a standard adder.

The corresponding rules for a Dadda multiplier are similar in concept to the Wallace multiplier except the reduction stages are more complicated; that is, we replace the second step above with the following:

- Combine any three wires with same weight using a full-adder; the result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
- If there are two wires with the same weight left, let w be that weight and then:
  - If $w \equiv 2 \bmod 3$ then combine the wires using a half-adder; the result in next stage is one wire of the same weight (i.e., the sum) and one wire a higher weight (i.e., the carry).
  - Otherwise, just pass them through to the next stage.
- If there is one wire with the same weight left, just pass it through to the next stage.

Consider a case were we are dealing with 4-bit inputs x and y; Table 7.1 demonstrates the actions applied in constructing the various stages of the Wallace and Dadda multipliers. Since they are similar, we consider only the Wallace case more carefully. The first stage multiplies each x_i with each y_j; we have one weight-0 wire from $x_0 \cdot y_0$ but three weight-4 wires from $x_1 \cdot y_3$, $x_3 \cdot y_1$ and $x_2 \cdot y_2$ for example. Following the first stage, we apply two reduction stages according to the rules. For example, in the first reduction stage there is one weight-1 wire as input so we use a pass-through (PT) operation which results in one weight-1 wire as output; there are two weight-2 wires so we use a half-adder (HA) operation which results in one weight-2 wire and one weight-4 wire as output; there are three weight-4 wires so we use a full-adder (FA) operation which results in one weight-4 wire and one weight-8 wire as output. Finally, we are left with the output of the second reduction stage in which all weights have two or less wires in them; we group the wires into two 8-bit values and add them using, for example, a ripple-carry adder. This might all sound a bit abstract; consider a concrete example where we want to multiply $x = 0110_{(2)} = 6_{(10)}$ and $y = 1010_{(2)} = 10_{(10)}$. Figure 7.11 shows the structure of the entire Wallace multiplier and the intermediate results produced. The final stage computes the result $00111100_{(2)} = 60_{(10)}$ as the sum of $00111100_{(2)}$ and $00000000_{(2)}$ where zeros are inserted where there are not enough wires.

Notice that the Wallace multiplier used six half-adders and four full-adders and the final addition has 6-bit inputs; the Dadda multiplier used one half-adder and five full-adders but the final addition has 5-bit inputs. As such, one can view the two approaches as a trade-off between complexity in the reduction layers against complexity of the (single) final addition. In each structure, there are $O(\log n)$ layers of reduction plus the initial and final layer. However, notice that within each reduction stage each half-adder or full-adder can operate independently from the rest. That is, each reduction stage has an $O(1)$ critical path which is why we see an improvement versus the original tree multiplier: the critical path is essentially through only $O(\log_2 n)$ full-adders rather than $O(n \log_2 n)$.

7.5.6 Booth Recoding

The technique of **recoding** either the multiplier or multiplicand to allow more efficient multiplication can be attractive: essentially we do some extra work before executing the multiplication so that the actual operation takes less steps. The **Booth encoding** technique [5] is a popular example of this; it takes advantage of repeated sequences of 1 or 0 digits in the multiplier. Using the recoding strategy, addition operations are only required at the boundaries of such sequences; no additions are required within the sequences themselves.

For example, consider the 8-bit multiplier value $30_{(10)} = 00011110_{(2)}$ where one can see there is a repeated sequence of four 1 digits in the binary representation. Where such a sequence exists from bits position i to bit position j, we can treat the sequence as a single digit of weight $2^{j+1} - 2^i$. So in this case we re-write the digits from $i = 1$ to $j = 4$ as a single digit of weight $2^{4+1} - 2^1 = 2^5 - 2^1 = 30$. Since we have recoded a long sequence as a single digit, a generic multiplication algorithm can take less steps: it requires less additions of partial products resulting from the combination of multiplier digits and the multiplicand.

Consider the example of multiplying $x = 6_{(10)} = 00000110_{(2)}$ by $y = 30_{(10)} = 00011110_{(2)}$. Using the method introduced previously, we compute the result as

0 0 0 0 0 1 1 0	input x
0 0 0 1 1 1 1 0	input y
0 0 0 0 0 0 0 0	partial product for $+x \cdot 0$
0 0 0 0 0 1 1 0	partial product for $+x \cdot 10$
0 0 0 0 0 1 1 0	partial product for $+x \cdot 100$
0 0 0 0 0 1 1 0	partial product for $+x \cdot 1000$
0 0 0 0 0 1 1 0	partial product for $+x \cdot 10000$
0 0 0 0 0 0 0 0	partial product for $+x \cdot 000000$
0 0 0 0 0 0 0 0	partial product for $+x \cdot 0000000$
0 0 0 0 0 0 0 0	partial product for $+x \cdot 00000000$
0 0 0 0 0 0 0 1 0 1 1 0 1 0 0	result $z = x \cdot y$

If we first recode y using the Booth method, the result is instead calculated as follows:

							0	0	0	0	0	1	1	0	input x
							0	0	0	1	1	1	1	0	input y
							0	0	+1	0	0	0	−1	0	recoded y
							0	0	0	0	0	0	0	0	partial product for $+x \cdot 0$
1	1	1	1	1	1	1	1	1	1	1	0	1	0		partial product for $-x \cdot 10$
				0	0	0	0	0	0	0	0	0			partial product for $+x \cdot 000$
			0	0	0	0	0	0	0	0	0				partial product for $+x \cdot 0000$
		0	0	0	0	0	0	0	0	0					partial product for $+x \cdot 00000$
	0	0	0	0	0	0	1	1	0						partial product for $+x \cdot 100000$
0	0	0	0	0	0	0	0	0							partial product for $+x \cdot 0000000$
0	0	0	0	0	0	0	0								partial product for $+x \cdot 00000000$
0	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	result $z = x \cdot y$

which requires less addition operations, i.e., additions of non-zero partial products, during accumulation of the partial products and hence potentially less time.

The problem with this approach is that in the worse case, it actually causes us to require *more* addition operations during accumulation of the partial products than the standard method. To show this, consider a multiplier $y = 5_{(10)} = 00000101_{(2)}$. Using the standard multiplication method we would require two addition operations, one for each 1 digit in y. Using a Booth recoding of y, we actually perform four additions:

							0	0	0	0	0	1	1	0	input x
							0	0	0	0	0	1	0	1	input y
							0	0	0	0	+1	−1	+1	−1	recoded y
1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	partial product for $-x \cdot 1$
					0	0	0	0	0	0	1	1	0		partial product for $+x \cdot 10$
1	1	1	1	1	1	1	1	1	1	1	0	1	0			partial product for $-x \cdot 100$
			0	0	0	0	0	0	1	1	0					partial product for $+x \cdot 1000$
		0	0	0	0	0	0	0	0	0						partial product for $+x \cdot 00000$
	0	0	0	0	0	0	0	0	0							partial product for $+x \cdot 000000$
0	0	0	0	0	0	0	0	0								partial product for $+x \cdot 0000000$
0	0	0	0	0	0	0	0									partial product for $+x \cdot 00000000$
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	result $z = x \cdot y$

which is clearly worse not better ! To combat this worst case and exploit the fact that the recoded multiplier implies less steps in multiplication, we use a further recoding strategy which is often termed the **modified Booth recoding**. The basic idea is to group the recoded digits into pairs (reading them from right to left). Within a pair (y_{i+1}, y_i) grouped from the recoded value y, the digit y_{i+1} has twice the weight of y_i so one can think of the pair as having the weight $2y_{i+1} + y_i$. This allows us to think of the pair $(+1, -1)$ as $2 - 1 = 1$ for example. Seven such pairs are possible, with the other two impossible due to the original Booth recoding technique; the pairs contribute the following weights:

$$
\begin{array}{llll}
y_{i+1} = 0 & \text{and} & y_i = 0 & \rightarrow \quad 0 \\
y_{i+1} = 0 & \text{and} & y_i = +1 & \rightarrow \quad +1 \\
y_{i+1} = 0 & \text{and} & y_i = -1 & \rightarrow \quad -1 \\
y_{i+1} = +1 & \text{and} & y_i = 0 & \rightarrow \quad +2 \\
y_{i+1} = +1 & \text{and} & y_i = +1 & \rightarrow \quad \text{not possible} \\
y_{i+1} = +1 & \text{and} & y_i = -1 & \rightarrow \quad +1 \\
y_{i+1} = -1 & \text{and} & y_i = 0 & \rightarrow \quad -2 \\
y_{i+1} = -1 & \text{and} & y_i = +1 & \rightarrow \quad -1 \\
y_{i+1} = -1 & \text{and} & y_i = -1 & \rightarrow \quad \text{not possible}
\end{array}
$$

Thus, using these pairings we can re-write our worst case as

$$
\begin{array}{cccccccccccccccc}
 & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{input } x \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{input } y \\
 & & & & & 0 & 0 & 0 & 0 & +1 & -1 & +1 & -1 & \text{recoded } y \\
 & & & & & & 0 & & 0 & & +1 & & +1 & \text{grouped } y \\
\hline
 & & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & \text{partial product for } +x \cdot 1 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & & \text{partial product for } +x \cdot 100 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \text{partial product for } +x \cdot 00000 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \text{partial product for } +x \cdot 0000000 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \text{result } z = x \cdot y
\end{array}
$$

which is no worse than using the standard method and makes explicit the fact that there are less steps: we only need to consider half the number of recoded multiplier digits versus the original.

7.6 Putting It All Together

To flesh out the components in the basic Verilog processor model developed at the end of Chapter 5, we need ALU modules to perform comparison and arithmetic on integer inputs. Previously we assumed that such operations could be created for us by the Verilog tool-chain, for example we used statements such as

```
alu_dst_lo = GPR[dec_rs] + GPR[dec_rt]
```

to model the behaviour of an addition instruction. In reality, we would like to conglomerate all such operations within the ALU so that we can minimise the amount of replicated logic and modularise the overall processor design.

7.6.1 Comparison ALU

The ALU for comparisons is the simplest to construct. We already have modules that perform unsigned equality and less than comparisons; our task is to use and pack-

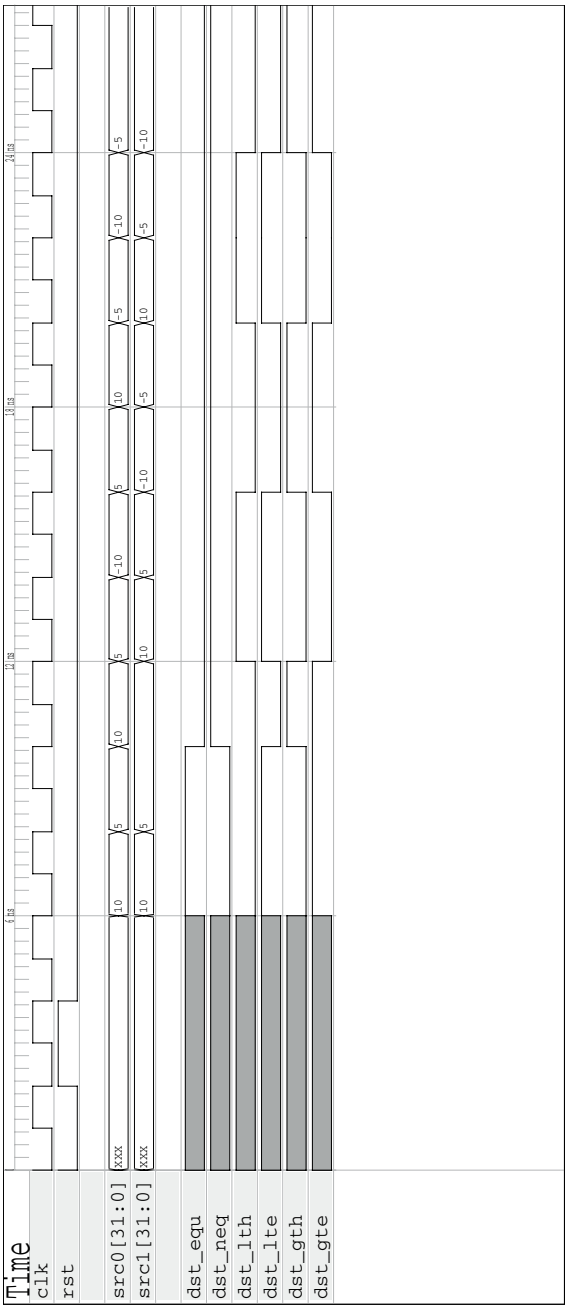


Figure 7.12 Behaviour of the ALU for comparisons.

```
module alu_cmp( input  wire        clk,
                input  wire        rst,

                input  wire [31:0] src0,
                input  wire [31:0] src1,

                output wire        dst_equ,
                output wire        dst_neq,
                output wire        dst_lth,
                output wire        dst_lte,
                output wire        dst_gth,
                output wire        dst_gte );

  wire w0;
  wire w1;
  wire w2;

  equ_nbit t0( .r(w0), .x(src0[31:0]), .y(src1[31:0]) );
  lth_nbit t1( .r(w1), .x(src0[30:0]), .y(src1[30:0]) );

  defparam t0.n = 32;
  defparam t1.n = 31;

  assign w2 = (  src0[31] &  src1[31] & ~w0 & w1 ) |
              (  src0[31] & ~src1[31]            ) |
              ( ~src0[31] & ~src1[31] & ~w0 & w1 ) ;

  assign dst_equ =  w0;
  assign dst_lth =  w2;

  assign dst_neq = ~dst_equ ;
  assign dst_lte =  dst_lth |
                    dst_equ ;
  assign dst_gth = ~dst_lte ;
  assign dst_gte = ~dst_lth ;

endmodule
```

Listing 7.9 A Verilog module that models an ALU for comparisons.

age these modules so they perform all variants of signed comparison. The Verilog module is shown in Listing 7.9. We have the ALU generate all possible comparison outputs continuously from the inputs `src0` and `src1`. That is, there is no need to pass the ALU an opcode: the processor simply sets the inputs and then bases any decision on the right comparison output. For example, if it wants to test if the values are equal then it examines the `dst_equ` output. This design is motivated by the fact that the MIPS32 instructions for conditional branches do not use the `funct` field of the R-type instruction encoding to determine the ALU operation like, for example, an addition would. Branches use the I-type encoding and so the operation performed is derived just from the `opcode` field.

The slight complication then is simply that we need signed comparison from our basic unsigned modules. We achieve this using a slight of hand. Firstly, note that signed equality is the same as unsigned equality: we just test if each bit is the same or not. Signed less than is the problem; to solve it we adopt the simple rules presented previously. That is, because of the way the twos-complement representation works we have that

```
module alu_exe( input  wire        clk,
                input  wire        rst,

                input  wire [ 5:0] oper,

                input  wire [31:0] src0,
                input  wire [31:0] src1,

                output wire [31:0] dst_lo,
                output wire [31:0] dst_hi );

  ...

endmodule
```

Listing 7.10 The interface to a Verilog module that models an ALU for arithmetic.

1. If `src0` is negative and `src1` is negative, they are not equal, and an unsigned less than comparison is true, then `src0` is less than `src1`.
2. If `src0` is negative and `src1` is positive we know `src0` is less than `src1`.
3. If `src0` is positive and `src1` is positive, they are not equal, and an unsigned less than comparison is true, then `src0` is less than `src1`.

Testing the sign of `src0` and `src1` is simply a matter of examining the most-significant bit. Using this fact we can apply our rules, and an unsigned comparison on the remaining 31-bit unsigned value, to generate the correct signed comparison. Then, since we have equality and less than, we construct all other comparisons from these. For example, `src0` is greater than `src1` if `src0` is not less than or equal to `src1`. The behaviour of the result is shown in Figure 7.12.

7.6.2 Arithmetic ALU

The ALU for arithmetic is somewhat more complex, partly because we have more operations and sub-module instances and partly because some of the operations are multi-cycle rather than continuous. The latter difference requires that we manage the clock and reset signals for the multiplier and shifter modules so they operate correctly. The Verilog code is too long to present on one page but the interface is shown in Listing 7.10. The general idea is that the ALU perform an operation, using `src0` and `src1` as inputs, which is determined by the `oper` input. The output is supplied in low and high parts via `dst_lo` and `dst_hi` which allows for both 32-bit outputs, for example addition, and 64-bit outputs, for example multiplication. The design does not include the facility to manage carry or overflow flags.

Listing 7.11 details the instantiation of the main sub-modules within the ALU. Instances `t3`, `t4` and `t5` are the addition/subtraction, multiplier and shifter modules described previously. Adder instances `t0`, `t1` and `t2` are used to manage the multiplier by ensuring it is supplied only with positive inputs. To achieve this, we negate `src0` and `src1` to produce `n0` and `n1` and select whichever one is pos-

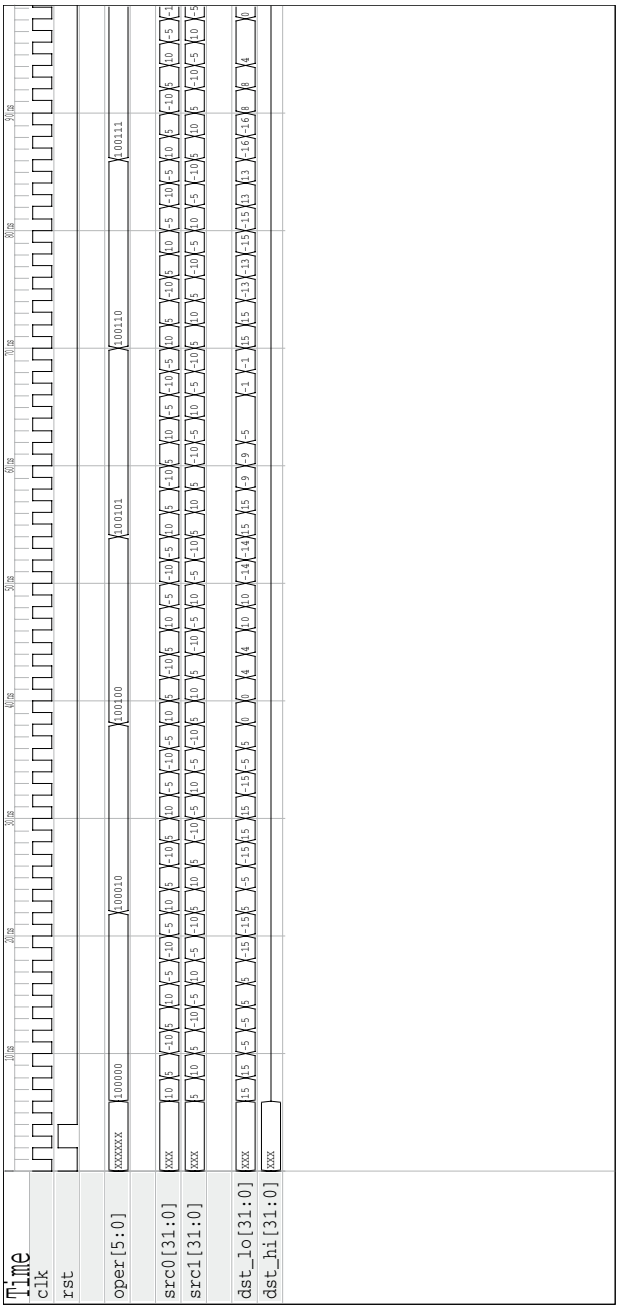


Figure 7.13 Behaviour of the ALU for arithmetic.

```
1  add_nbit t0(.ci(1'b0),.r(n0),.x(~src0),.y(32'b1));
2  add_nbit t1(.ci(1'b0),.r(n1),.x(~src1),.y(32'b1));
3  add_nbit t2(.ci(1'b0),.r(n2),.x(~  w1),.y(64'b1));
4
5  defparam t0.n = 32;
6  defparam t1.n = 32;
7  defparam t2.n = 64;
8
9  assign n3 = src0[31]             ? n0 : src0;
10 assign n4 = src1[31]             ? n1 : src1;
11 assign n5 = src0[31] != src1[31] ? n2 :   w1;
12
13 assign ci = 0;
14
15 sub_nbit t3( .ctrl(     s0),
16
17             .ci  (     ci),
18             .co  (     co),
19             .x   (   src0),
20             .y   (   src1),
21             .r   (     w0) );
22
23 mul_nbit t4( .clk (    clk),
24             .rst (mul_rst),
25
26             .x   (     n3),
27             .y   (     n4),
28             .r   (     w1) );
29
30 shf_nbit t5( .clk (    clk),
31             .rst (shf_rst),
32
33             .oper(     s1),
34             .x   (   src0),
35             .y   (   src1),
36             .r   (     w2) );
37
38 defparam t3.n = 32;
39 defparam t4.n = 32;
40 defparam t4.m =  5;
41 defparam t5.n = 32;
42 defparam t5.m =  5;
```

Listing 7.11 Verilog code to instantiate internal sub-modules.

itive to produce `n3` and `n4`; these are fed to the multiplier. The multiplier output is patched up to generate the right sign by comparing the input signs and selecting either the actual multiplier output `w1` or the negation `n2` to produce the final result `n5`. Note also that we tie the addition/subtraction carry-in to 0 although in reality this value might come from a carry flag stored in the processor status register.

Listing 7.12 details the generation of the basic logic operations and the two multiplexers used to select the result from all those possible, which is then fed to the ALU output. Notice that the multiplexers are controlled by the `s3` signal; this is managed by two processes shown in Listing 7.13. The second process is the most interesting: whenever the `oper` signal changes, i.e., we want to perform a new operation, a case statement is executed which compares the opcode to the list of possibilities and sets the right control signals in each case. The multiplier and shifter cases are the most

```
genvar i;
generate
  for( i = 0; i < 32; i = i + 1 )
  begin:gen_alu_exe
          and t6( w3[i], src0[i], src1[i] );
           or t7( w4[i], src0[i], src1[i] );
          xor t8( w5[i], src0[i], src1[i] );
          nor t9( w6[i], src0[i], src1[i] );

    mux8_1bit ta( .r(dst_lo[i]), .i0(w0[i]), .i1(n5[i+ 0]),
                                 .i2(w2[i]), .i3(w3[i   ]),
                                 .i4(w4[i]), .i5(w5[i   ]),
                                 .i6(w6[i]), .i7(1'bZ    ),

                                 .s0(s2[0]), .s1(s2[1]), .s2(s2[2]) );

    mux8_1bit tb( .r(dst_hi[i]), .i0(1'bZ ), .i1(n5[i+32]),
                                 .i2(1'bZ ), .i3(1'bZ    ),
                                 .i4(1'bZ ), .i5(1'bZ    ),
                                 .i6(1'bZ ), .i7(1'bZ    ),

                                 .s0(s2[0]), .s1(s2[1]), .s2(s2[2]) );
  end
endgenerate
```

Listing 7.12 Verilog code to generate logic operations and multiplexers.

involved; as well as selecting the right multiplexer control signal, these need to reset the associated device to clear the internal state and initiate a new operation. The first process ensures this reset signal is cleared soon afterwards ready for the next operation to be started. The behaviour of the result is shown in Figure 7.13.

7.7 Further Reading

- M.D. Ercegovac and T. Lang.
  Digital Arithmetic.
  Morgan Kaufmann, 2003. ISBN: 1-558-60798-6.
- M.J. Flynn and S.F. Oberman.
  Advanced Computer Arithmetic Design.
  John Wiley, 2001. ISBN: 0-471-41209-0.
- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.
- I. Koren.
  Computer Arithmetic Algorithms.
  A.K. Peters, 2001. ISBN: 1-568-81160-8.
- J.E. Stine Jr.
  Digital Computer Arithmetic Datapath Design Using Verilog HDL.
  Kluwer, 2003. ISBN: 1-402-07710-6.

```
always @ ( negedge clk )
begin
  shf_rst = 1'b0;
  mul_rst = 1'b0;
end

always @ ( oper )
begin
  case( oper )
    `FUNCT_ADD  : begin
                    s0 = 0;
                    s2 = 0;
                  end
    `FUNCT_SUB  : begin
                    s0 = 1;
                    s2 = 0;
                  end
    `FUNCT_MUL  : begin
                    mul_rst = 1;
                    s2      = 1;
                  end
    `FUNCT_MULT : begin
                    mul_rst = 1;
                    s2      = 1;
                  end

    `FUNCT_AND  : s2 = 3;
    `FUNCT_OR   : s2 = 4;
    `FUNCT_XOR  : s2 = 5;
    `FUNCT_NOR  : s2 = 6;

    `FUNCT_SLLV : begin
                    shf_rst = 1;
                    s1      = `SHF_L_LOGIC;
                    s2      = 2;
                  end
    `FUNCT_SRLV : begin
                    shf_rst = 1;
                    s1      = `SHF_R_LOGIC;
                    s2      = 2;
                  end
  endcase
end
```

Listing 7.13 Verilog processes that determine the ALU behaviour.

7.8 Example Questions

29. a. Comparison operations for a given processor take two 16-bit operands and return zero if the comparison is false or non-zero if it is true. By constructing some of the comparisons using combinations of other operations, show that implementing all of $=, \neq, <, \leq, >$ and $\geq$ is wasteful. State the smallest set of comparisons that need dedicated hardware such that all the standard comparisons can be executed.

b. The ALU in the same processor design does not include a multiply instruction. So that programmers can still multiply numbers, write an efficient C function to

multiply two 16-bit inputs together and return the 16-bit lower half of the result. You can assume the inputs are always positive.

c. A population count operation takes an n-bit value x and computes the Hamming weight $H(x)$, i.e., the number of bits in x which are set to 1. Some processors have a dedicated instruction to do this but the proposed one does not; write an efficient C function to compute the population count of 16-bit inputs.

30. Imagine we want to compute the result of multiplying two n-bit numbers x and y together, i.e., $r = x \cdot y$, where n is even. One can adopt a divide-and-conquer approach to this computation by splitting x and y into two parts each of size $n/2$ bits

$$\begin{aligned} x &= x_1 \cdot 2^{n/2} + x_0 \\ y &= y_1 \cdot 2^{n/2} + y_0 \end{aligned}$$

and then computing the full result

$$r = r_2 \cdot 2^n + r_1 \cdot 2^{n/2} + r_0$$

via the parts

$$\begin{aligned} r_2 &= x_1 \cdot y_1 \\ r_1 &= x_1 \cdot y_0 + x_0 \cdot y_1 \\ r_0 &= x_0 \cdot y_0. \end{aligned}$$

The naive approach above uses four multiplications of $(n/2)$-bit values. The Karatsuba-Ofman method reduces this to three multiplications (and some extra low-cost operations); show how this is achieved.

31. Assume we are using unsigned integers represented in 4 bits.

a. What is the result of using a normal 4-bit adder circuit to compute the sum $10 + 12$?

b. A saturating or clamped adder is such that if an overflow occurs, that is the result does not fit into 4 bits, the highest possible number is returned as a result. With a clamped 4-bit addition denoted by $\uplus$, we have that $10 \uplus 12 = 15$. In general, for an n-bit clamped adder

$$x \uplus y = \begin{cases} x + y & \text{if } x + y < 2^n \\ 2^n - 1 & \text{otherwise} \end{cases}$$

Design a circuit that implements a 4-bit adder of this type.

32. Give a 1-line C expression to test if a non-zero integer x is an exact power-of-two; i.e., if $x = 2^n$ for some n then the expression should evaluate to a non-zero value, otherwise it evaluates to zero.

Chapter 8
Memory and Storage

640K [of main memory] ought to be enough for anybody.

– W.H. Gates [alleged, almost certainly misquoted, but still amusing]

Abstract In Chapter 5 we skirted around the idea of long-term memory much like we did with the idea of computation and the ALU. This was an attractive option at the time because it meant we could focus on the processor itself, but the concept of memory and storage is of course central to the operation of the processor: everything it does is oriented toward computation on (large numbers of) values we must store somewhere if the program being executed is not trivial. To examine these issue in more detail, in this chapter we investigate the design and implementation of a range of memory and storage devices focusing on those most pertinent to processor design. Specifically, we include registers and main memory already discussed previously, describing in basic terms how they can be modelled in Verilog. Ultimately, our goal is to build a memory hierarchy capable of supplying data and instructions to our processor data-path via a standard mechanism; to achieve this efficiently and robustly we include an investigation of cache memory and issues such as error correction.

8.1 Introduction

A **memory**, or more generally a **storage device**, enables us to store items of data for a period of time. Such devices represent a crucial part of any computer system since, for example, we need to store values somewhere while the processor performs computation on them. However, even though the speed of processors has increased roughly in line with the increase in transistor performance, memory access time has improved much more slowly; only a few percent each year, about 30% in ten years. This is a massive problem: for example, it does not matter how fast we can perform computation in our ALU if loading and storing operands is slow. Furthermore, and perhaps more importantly, we have already seen that the fetch-decode-execute cycle

requires the processor to fetch instructions; we perform at least one load from the memory where instructions are stored in each processor cycle. If this fetch cannot be completed quickly, the entire fetch-decode-execute cycle will be dominated by the cost of accessing memory. This problem is often called the **memory wall** [68] or **von Neumann bottleneck** and it demands we, as computer architects, think carefully about all aspects of memory and storage device design.

Throughout this chapter, we try to refer only to memories rather than the more general case of storage devices which might include magnetic and optical disks for example. This might seem a bit odd but very roughly, only when discussing their implementation are the two different: abstractly we can follow Chapter 5 and describe either as an infinite array MEM in which we can store n-bit binary vectors. Two main functions are required to access the structure:

$$\begin{array}{ll} \textsc{Load}(x) & : \{0,1\}^* \to \{0,1\}^n, \quad \textsc{Load}(x) = MEM[x] \\ \textsc{Store}(x,y) & : \{0,1\}^* \times \{0,1\}^n \to \perp, \quad \textsc{Store}(x,y) = MEM[x] \leftarrow y \end{array}$$

In this context x is an **address** or **index**, a name or label used to identify a particular **element** or **location**. Given an address x, the LOAD function retrieves an element while the STORE function updates an element with the value y. Essentially, the main differences between what we might normally call memories and storage devices are just how we implement the LOAD and STORE functions in a concrete way, and exactly how addresses refer to particular elements.

We can categorise our memory or storage devices according to a number of key characteristics:

- A **short-term** or **volatile** memory retains stored data while the device is powered; when it is powered off, the content is lost. A **long-term** or **non-volatile** memory solves this problem by retaining stored data even after the device is powered off.
- A **Read-Write (RW)** or **mutable** memory can be used to overwrite or store new data items and retrieve existing items. The content of a **Read-Only (RO)** or **immutable** memory cannot be altered; one can only retrieve existing data items. Read-Only memories are often abbreviated **ROMs**.
- When using a **random-access** memory, one can store or retrieve *any* data item; one is not restricted in terms of the order the data items can be accessed in. A **sequential-access** memory imposes restrictions on the *order* data items can be accessed in; one must access them in sequence. Random-access memories are often abbreviated **RAMs**.
- A **primary** memory is connected directly to the processor; usually such a device will be used to store active items of data which are required for the current task. A **secondary** memory does not usually have a direct connection to the processor, it is accessed as a peripheral via some other interface.

Thus far we have talked about memory being used to store and retrieve operands for the ALU. In order for the ALU to operate however, we need both operands and operators: the values that decide which arithmetic function the ALU performs. Thus we can place stored content into one of two classes: data, which are the values we

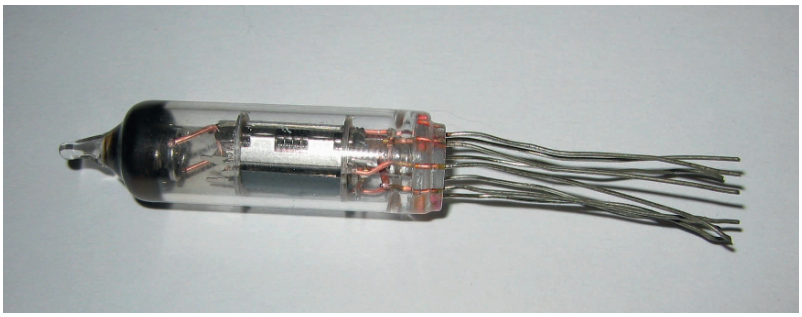


Figure 8.1 A military quality sub-miniature vacuum tube. (Reproduced by permission under the GNU Free Documentation License, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:CV4501.JPG`)

operate on, and instructions, which tell us which operation to perform. Given this definition, and recapping for completeness on the discussion in Chapter 4, one can reasonably construct two distinct forms of memory organisation:

Harvard Architecture A Harvard architecture treats data and instructions as separate entities. One would expect two separate memory hierarchies, and hence two interfaces to said hierarchies, dedicated to storing exclusively *either* data *or* instructions.

von Neumann Architecture A von Neumann architecture takes the opposite approach. There is no inherent difference between data and instructions; we have already seen that both are just numbers, it depends on how you interpret those numbers. For example, the number 12 might represent the integer value twelve if used as an ALU operand or represent the code used to invoke the addition function if used as an ALU operator. Thus a von Neumann architecture has a single, unified memory hierarchy that is used to store *both* data *and* instructions in the same way.

8.1.1 Historical Memory and Storage

In Chapter 4 we introduced EDVAC, the first practical stored program computer, which was based on the stored program architecture. EDVAC is typical of even modern computers in that it used three main forms of storage device: a **vacuum tube**-based accumulator to store very short term data that resulted from computation by the ALU, a **mercury delay line**-based volatile memory for longer term data and instructions, and a **paper tape** reader for non-volatile, long-term storage.

The vacuum tube was the technology replaced by smaller, more reliable devices such as transistors. Essentially they perform the same function, but are realised us-

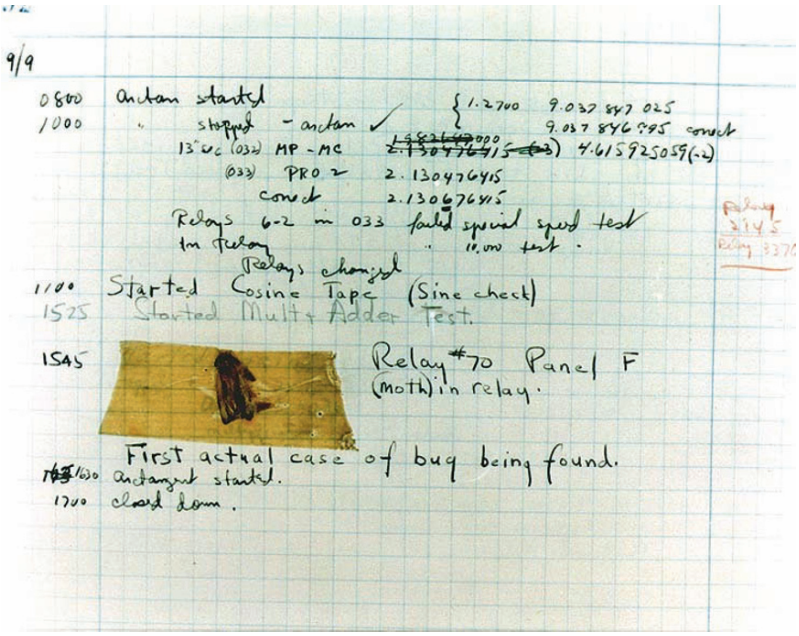

9/9
0800 Arctan started
1000 " stopped - arctan ✓
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

Figure 8.2 The infamous Harvard Mark II "bug" ! (A U.S. Navy photograph now in public domain, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:H96566k.jpg`)

ing a different technology; as such they can be used to construct logic and storage devices. A typical vacuum tube, or thermionic valve, looks very much like a light bulb: it uses a glass or ceramic envelope to hold a vacuum that surrounds an electron-producing filament (or cathode) and a metal plate (or anode). When the filament is heated, electrons are produced into the vacuum which are attracted by the plate resulting in a current between the two. In simple terms, this implements a basic switch: when the filament is heated the switch is on, when it is cooled the switch is off. Although similar logic and storage devices can be constructed using a vacuum tube as one would construct using a transistor, the scale of such devices is limited by reliability. Specifically, the filament slowly degrades due to impurities in the tube; the vacuum in the envelope becomes less pure as time progresses. Furthermore, the filament is stressed during the heating process which will eventually lead to it burning out; early users realised that this happened particularly often during powering up or down of a vacuum tube-based device. Although cooling devices eased this problem, tube failure was one of the most common causes of early computer crashes.

In fact, the terms "bug" and "debug" in relation to programming both stem (at least in part) from failure of this sort. In 1945, programmers of the Harvard Mark II computer developed by Howard Aiken, discovered a moth inside one of the com-

ponents; the unfortunate insect had shorted the component which had resulted in the computer malfunctioning. Although the terms had been used in various areas of engineering previously, Grace Hopper and her programmers are often cited as introducing them in the context of computer science. Certainly this real-life bug is so famous that it is still on display in the Smithsonian Museum of American History !

A mercury delay line is essentially a tube filled with liquid mercury with a pair of microphones and a speaker at each end. To store a value, the speaker at one end sends a signal representing the value into the mercury tube; due to the high speed of sound in mercury, the signal then travels along the tube faster than in a medium such as air. Upon receiving the signal at the other end via the microphone, it was relayed to the speaker and sent down in the opposite direction. This process was repeated, effectively storing the value as a perpetually regenerated looping signal within the tube. To read the value, one simply intercepted the signal as it was received by one of the microphones. To update the value, one again intercepted the signal as it was received by one of the microphones but replaced it with the new value before passing it on to the speaker for reinsertion into the tube. EDSAC stored 512 words of memory within thirty two delay lines. As one can imagine, the cost and toxicity of mercury presented a problem in terms of their use. Other problems were presented by the speed of sound in mercury, which limited the number of bits stored by each tube to around 576, and the fact that this speed changed as the mercury temperature altered ! Even so, the devices were much more reliable than other technologies of the time and have been used in computers as recently as the 1960s.

Although Babbage had proposed to program his Analytical Engine via a system of paper cards, perhaps the first use of punched card storage was by inventor Joseph Marie Jacquard in the early 1800s. Jacquard developed a weaving machine or loom whose operation, i.e., the pattern of weaving, was determined by a primitive program encoded as punched holes in a card. Roughly speaking, each hole in the card controlled a needle in the machine, each card represented one row of the intended design. The weaving process was therefore simplified from manual configuration of the machine, to semi-automatic weaving whereby the operator simply swapped in the right card at the right time. Later, in 1890 Herman Hollerith used a similar storage medium to encode US census data. He developed a mechanical tabulating machine which was able to process the cards, each of which represented the details of an individual, and produce statistics at an unprecedented speed: the 1890 census was processed in less than a quarter of the time the 1880 census had taken to complete by hand. In 1896 Hollerith founded the Tabulating Machine Company, an early incarnation of the company we now know as IBM, to exploit his invention.

Paper tape storage is a fairly natural progression from punched cards and the two mediums share much of the same technology. Instead of individual cards, paper tape is a continuous ream of paper which is notionally divided up into rows. Each row allowed for a number of holes to be punched; early tape systems allowed five holes which allowed for the use of Baudot character encodings, later systems allowed up to eight which allowed for more modern encodings such as ASCII. The tape would have perforated sprocket holes on the edge which allowed it to be fed through a tape reader which, using either a mechanical or optical mechanism, was able to translate

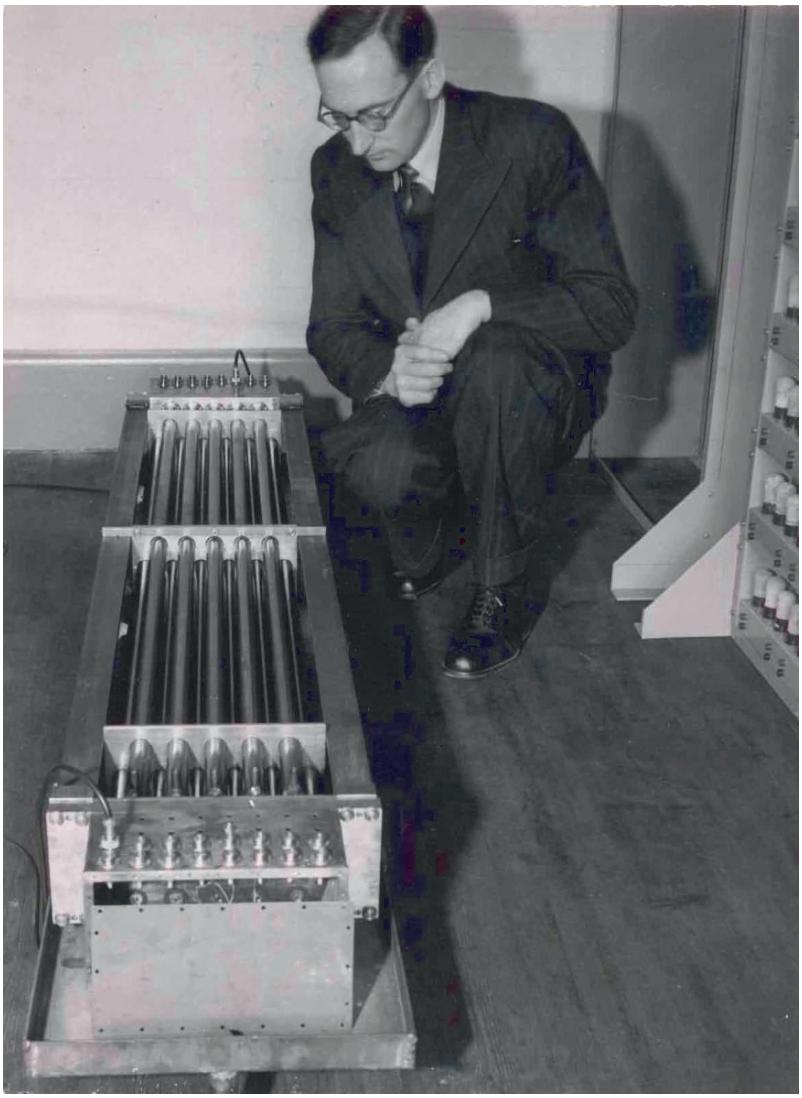


Figure 8.3 Maurice Wilkes with an EDSAC mercury delay line. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/delay_lines.jpg`)

Figure 8.4 Experimental EDSAC memory tank. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/edsac99.23.jpg`)

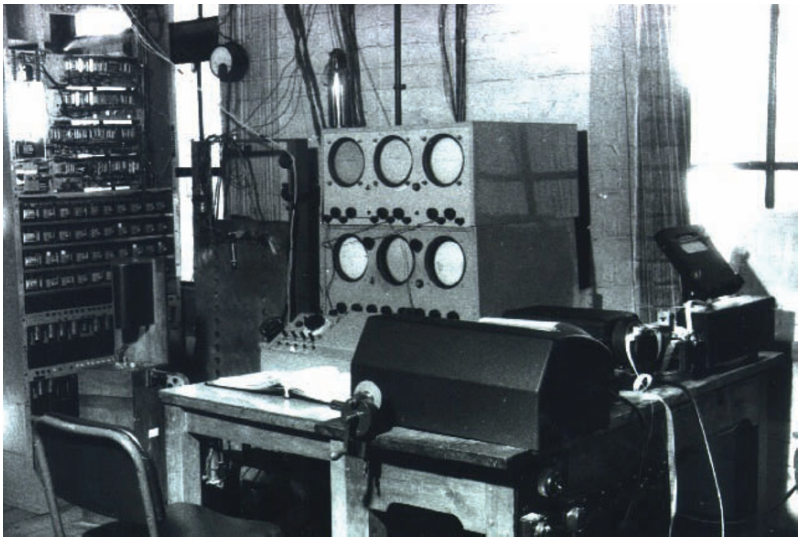


Figure 8.5 Experimental EDSAC magnetic tape rack. (Copyright Computer Laboratory, University of Cambridge. Reproduced by permission under the Creative Commons Attribution Licence, photographer: Unknown, source: `http://www.cl.cam.ac.uk/Relics/jpegs/edsac99.19.jpg`)

Figure 8.6 A fanfold DEC paper tape containing a PDP-11 test program. (Reproduced by permission under the GNU Free Documentation License, photographer: Unknown, source: `http://en.wikipedia.org/wiki/Image:Papertape2.jpg`)

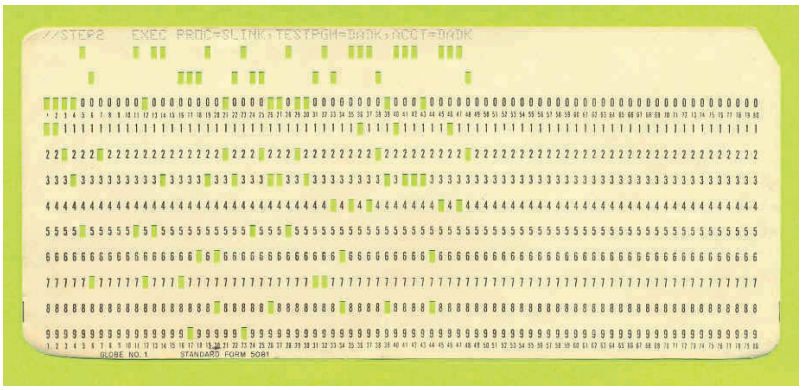


Figure 8.7 An IBM 5081 data processing card containing a line of DOS JCL program. (Reproduced by permission under the GNU Free Documentation License, photographer: Dick Kutz, source: `http://en.wikipedia.org/wiki/Image:Punch-card-5081.jpg`)

each row of holes into a binary word representing a character. Although the tape could be prepared off-line by the operator (encoding either a program of instructions or simply data values) and read at high speed, two main problems existed. Firstly, the tape was somewhat unreliable in the sense that it was prone to damage and for holes to be unreadable due to inaccuracies in the punching mechanism. Secondly, the tape mechanism was difficult to rewind; the paper tape storage device could thus be described as sequential-access only.

8.1.2 A Modern Memory Hierarchy

The previous discussion of historical memory and storage devices within the EDVAC computer was no accident; the three different levels of device live on today in modern computers although implemented using more modern technologies. The different levels form what is commonly termed the **memory hierarchy** or **storage hierarchy**, a tiered description of how the different levels compare to and interact with each other. The different levels of the memory hierarchy are managed by different parts of the system; access to the registers is performed directly by a program executing on the processor, access to a disk is usually performed by the operating system on behalf of the program during execution.

Ideally we would like very large, very fast memories but because of physical constraints this is usually impossible. To approximate the ideal, we use a combination of technologies. Roughly speaking, as one reads from top to bottom of the pyramid in Figure 8.8 the devices get farther away from the processor, are able to store more elements, are slower to access, and are less expensive to build but larger

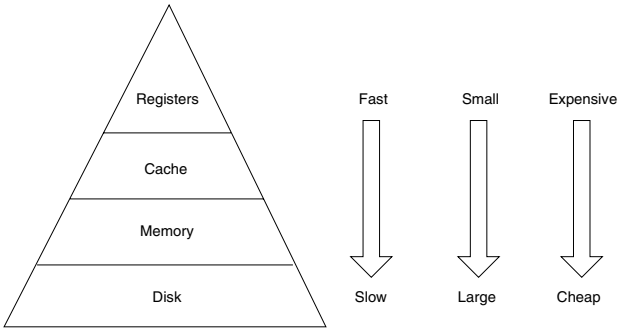
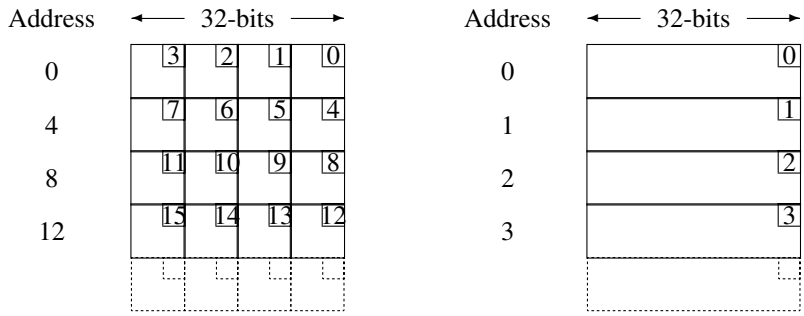


Figure 8.8 A modern memory hierarchy.

Level	1	2	3	4
Name	Registers	Cache	Memory	Disk
Size	$O(1\text{kB})$	$O(1\text{MB})$	$O(1\text{GB})$	$O(1\text{GB})$
Access Time	$O(1\text{ns})$	$O(10\text{ns})$	$O(100\text{ns})$	$O(1000\text{ns})$
Bandwidth	$O(10000\text{MB/s})$	$O(1000\text{MB/s})$	$O(100\text{MB/s})$	$O(10\text{MB/s})$
Cost	$O(100\$/\text{MB})$	$O(100\$/\text{MB})$	$O(10\$/\text{MB})$	$O(0.1\$/\text{MB})$
Managed By	Programmer	Processor	OS	OS

Table 8.1 A tabular description of typical price/performance characteristics for memory hierarchy components.

in physical size. The registers are at the top: they hold small-sized elements but are very quick to access. The next levels are the caches and main memory, which are larger but slower to access due to the fact that they are often implemented as a separate device to the processor. The next levels incorporate longer-term storage in the shape of magnetic and optical disks and tapes; these can hold the largest amount of information but are the slowest to access. The idea of course is that we should use a combination of all these devices, using the right one for the right job. Where we need to store information that need to be accessed quickly so we can feed operands to the ALU, we use a register. Where we need to store less transient information in larger quantities for the lifetime of the program, we use the main memory. Where we want to store vast amounts for information for periods of time which span when the device is powered, we use a disk or tape.



(a) 8-bit resolution (byte addressable). (b) 32-bit resolution (word addressable).

Figure 8.9 8-bit versus 32-bit memory resolution.

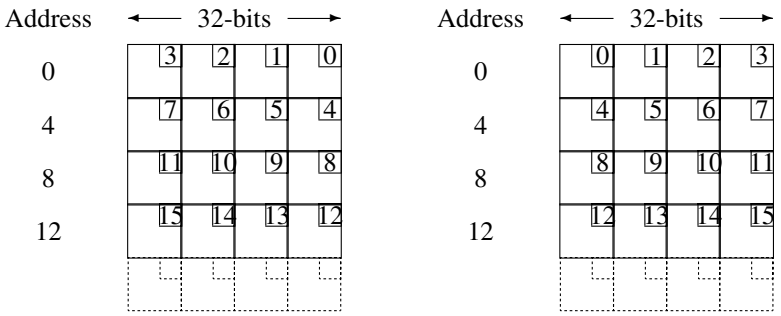
8.1.3 Basic Organisation and Implementation

8.1.3.1 Resolution

In reality we cannot build an infinitely large memory so the size of MEM must be set to say m elements; addresses used to access MEM are then bit-vectors of length $\lceil \log_2(m) \rceil$. Clearly there is a trade-off here: the larger the addresses we have, the more elements we can address for example. More exactly, if we have larger addresses we can either address many elements, each of which store elements of a small size, or fewer elements of a larger size. This relates to the **resolution** of the memory. Essentially the resolution dictates the size of each addressable element.

Consider an example using 10-bit addresses. We can clearly address 1024 byte sized elements in the memory, however we could also address 1024 word sized elements if we want. In the former case we have that the total memory size is 1024 bytes but, given 32-bit words, in the latter case the total size is 4096 bytes. Figure 8.9 attempts to demonstrate the difference; in each case memory elements are drawn as boxes with the address in the top-right corner, the elements are arranged in 32-bit groups with the starting address of the group to the left. In Figure 8.9a each 8-bit element has a distinct address and is individually accessible; in Figure 8.9b one can only access each 32-bit element.

Typical processors use **byte addressed** memory where each element is a byte; each byte has a unique address. So the operation $\textsc{Load}(x)$ retrieves the byte at address x and $\textsc{Store}(x, y)$ updates the byte at address x with value y.



(a) Little-endian ordering. (b) Big-endian ordering..

Figure 8.10 Little-endian versus big-endian memory ordering.

8.1.3.2 Endianness

Chapter 1 touched briefly on the problem of **endianness**, now we can see the problem in a more concrete setting. Suppose the memory *MEM* is byte addressable, i.e., holds 8-bit values, and we want to load a 32-bit word that starts at address 0. Clearly we would have to transfer four 8-bit bytes to construct this word; the question is, which are the most and least-significant bytes ? We could legitimately load the byte at address 0 and make this the least-significant continuing so that the byte at address 3 is the most-significant. As long as we were consistent, it would be equally as valid to do exactly the opposite: load the byte at address 0 and make this the most-significant continuing so that the byte at address 3 is the least-significant. The solution, which must be consistently enforced between load and store operations, is dictated by the endianness of the processor. The first choice means the processor is little-endian, the second choice means the processor is big-endian. Some processors can operate in both modes, one at a time, with their endian semantics selected by setting a flag in the status register when powered on. MIPS32 is an example of such a design; we will assume a little-endian system except where noted otherwise.

Again, Figure 8.10 attempts to demonstrate the difference; in each case memory elements are drawn as boxes with the address in the top-right corner, the elements are arranged in 32-bit groups with the starting address of the group to the left. In this case, Figure 8.10a has the 8-bit elements grouped in a little-endian order (the least-significant element is to the right, the most-significant to the left) while Figure 8.10b groups them in a big-endian order.

One can easily write a C function to test if a processor uses a little-endian or big-endian byte ordering; Listing 8.1 demonstrates this technique. The function places a 32-bit constant into a value shadowed by an array of four 8-bit bytes; we can then test which of the bytes in the value have been placed in which locations in the array and deduce the endian semantics as a result.

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char* argv[] )
{
  unsigned char x[4];
  unsigned int  y;

  *( unsigned int* )( x ) = 0x03020100;

  y   = ( ( unsigned int )( x[ 0 ] ) <<  0 ) |
        ( ( unsigned int )( x[ 1 ] ) <<  8 ) |
        ( ( unsigned int )( x[ 2 ] ) << 16 ) |
        ( ( unsigned int )( x[ 3 ] ) << 24 ) ;

  if( y == 0x03020100 )
    printf( "little endian\n" );
  else
    printf( "big endian\n" );
}
```

Listing 8.1 A short C program to test the endianness of a processor.

8.1.3.3 Alignment

Again suppose the memory *MEM* is byte addressable, i.e., holds 8-bit values, and that we want to load a 32-bit word. In theory, we can load the four bytes that would be used to construct our word from any address. For example, we might load $0, 1, 2$ and 3 meaning the address of the word was 0, or load $1, 2, 3$ and 4 meaning the address of the word was 1. However, in practise there is some difference between these two cases.

Where an address $x \equiv 0 \pmod{w}$ we say that it is w-aligned. Less formally, in our case above if x is a multiple of four, then we say that x is word-aligned. The address 0 is word-aligned as are 4, 8, 12 and so on; the address 1 is not word-aligned. Given that the word size is typically a power-of-two, and more exactly a power-of-two number of bytes, processors often make some distinction between **aligned** and **unaligned** addresses when used in load and store operations. An aligned address can usually be accessed more quickly than an unaligned address. Firstly, this is because the byte address of an aligned word address can more easily be calculated and vice versa: one simply adds or removes bits from the least-significant end of the address. For example, when we load a 32-bit word we can add two zero bits to the least-significant end of the word address to get us the byte address; word address 0 becomes byte address 0, word address 1 becomes byte address 4 and so on. Secondly, as we will see later, if the addresses are aligned it makes it much easier to design and build other elements of the memory hierarchy such as caches.

Sometimes the distinction is explicit; an ISA will include different instructions for aligned and unaligned memory access and cause an error if, for example, an unaligned address is used with an instruction expecting an aligned address. Sometimes the issue is implicit; the ISA only has a single set of access instructions that a micro-

```
module hilevel_mem( input  wire       clk,
                    input  wire       rst,

                    input  wire [7:0] addr,
                    input  wire [7:0] data_w,
                    output wire [7:0] data_r,
                    input  wire       mrq,
                    output reg        mrs,
                    input  wire       re,
                    input  wire       we );

  reg  [7:0] data[0:3];

  always @ ( posedge mrq )
  begin
    if     ( re )
      data_r = data[ addr ];
    else if( we )
      data[ addr ] = data_w;

    #6 mrs = 1;
       @( negedge mrq );
    #6 mrs = 0;
  end

endmodule
```

Listing 8.2 A Verilog module that implements a high-level, 4-element byte-addressable memory.

architecture may execute quicker or slower depending on the alignment properties of the address.

8.1.3.4 Implementation

Accesses by the processor to the memory via the address and data buses that connect them are carried out according to a protocol we will call a **memory transaction**; this is really a special case of a more general asynchronous bus protocol. Because it is asynchronous, the protocol cannot rely on the clock to keep the processor and memory in step with each other. Instead, they use several signals to implement a **handshake**, the memory side of which is implemented in Listing 8.2, whereby each device only progresses once the other is ready:

1. The processor sets the `addr` signal to the address in memory it wants to access. If it is performing a LOAD operation, it sets the read enable or `re` signal to high; if it is performing a STORE operation, it sets the write enable or `we` signal to high. Finally, it sets the memory request or `mrq` signal to high indicating to the memory that it wants to actually perform the access.
2. As soon as the memory notices the `mrq` signal is high, it starts the access. If `re` is high, it retrieves the value at address `addr` and drives this onto the output `data_r`; if `we` is high it sets the value at address `addr` to equal the input `data_w`. Completion of the actual operation will take some time; this is termed the **memory latency**. Once the operation is completed, the memory sets

the memory response signal or `mrs` to high to indicate to the processor it has finished.
3. As soon as the processor notices the `mrs` signal is high, it sets `mrq` to low to complete that side of the handshake. In the case of a LOAD operation, it stores the value passed to it on `data_r` into wherever it needs to.
4. As soon as the memory notices that the `mrq` signal is low, it sets `mrs` to low to complete that side of the handshake. Both processor and memory are now ready to continue performing more memory accesses or other operations.

The Verilog module uses a 2-dimensional register (confusingly also called a memory), to store the values. When the memory request signal `mrq` is raised, the access process is invoked: if the read enable or `re` signal is set, the `data_r` output is assigned to the value in memory addressed by `addr`; if the write enable or `we` signal is set, the value in memory addressed by `addr` is set to the input `data_w`. Once the operation is completed, the module introduces an artificial delay, to model the memory latency, before raising the memory response signal `mrs` to indicate the end of the transaction. The behaviour of the module is shown in Figure 8.11. In this case, the test stimulus performs two STORE operations to addresses 1 and 2 with the values 10 and 20. Notice how the `mrq` and `mrs` signals are used to synchronise and enforce an order to the transaction. Two LOAD operations are then used to read addresses 0 and 1; reading address 0 gives an undefined result since we have not yet stored anything in it, reading address 1 gives the value 10 which was previously stored.

Because the access speed, or latency, of the devices lower down the memory hierarchy is slower than those at the top, the **unit of transfer** is typically bigger as well. Note that this is not strictly related to the bus width: one might have to make several bus transactions to transfer a chunk of data, the unit of transfer relates to the size of the chunks requested at one time. For example, one might expect to read a single 32-bit value at a time from a register file, but a 4kB page at a time from a disk. By reading larger chunks at a time, the number of accesses will be smaller: this somewhat compensates for the overhead associated with a single access. Although we do not cover it in any depth, this issue is related to the concept of access via a **burst mode**. The idea is that several accesses close together in time, i.e., straight after each other, can be performed quicker than the same number of accesses performed at arbitrary points in time. Again, this compensates somewhat for the overhead associated with a single access, this time by sort of treating several accesses in a row as one.

The problem with this basic module is that it is a little too abstract in the sense that we have glossed over the issue of how the 2-dimensional register is actually constructed using known low-level components: a scalable design needs more careful consideration. Figure 8.13 details, in a rough sense, a way of constructing larger memories; it shows a 4-element memory which holds n-bit words (given the dashed boxes represent the logic equivalent of continuation dots). The idea is that each row of the main body of the diagram on the right-hand side is a group of n **memory cells** that each store a 1-bit value. Since we have already described how they work, here we use D-type latches for our memory cells; we will see later how memory

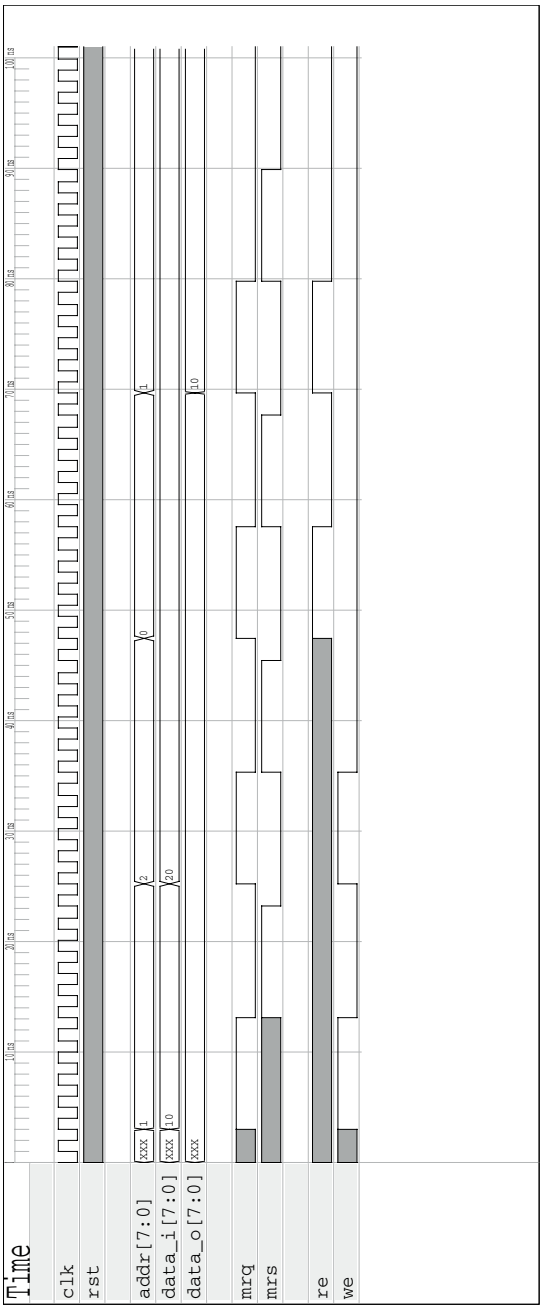


Figure 8.11 Behaviour of the high-level, 4-element, byte-addressable memory.

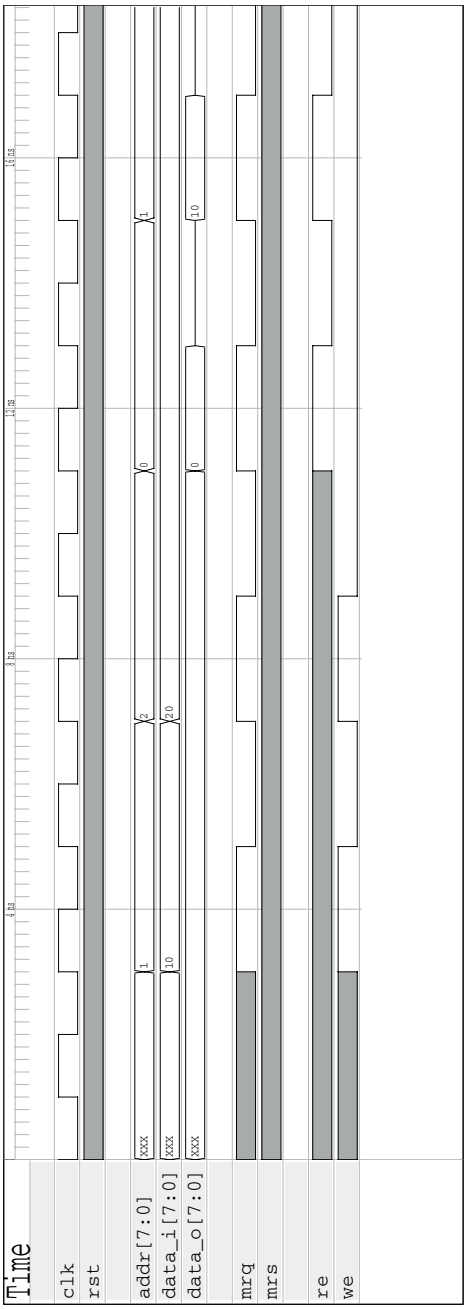


Figure 8.12 Behaviour of the low-level, 4-element, byte-addressable memory.

```
module lolevel_mem( input  wire       clk,
                    input  wire       rst,

                    input  wire [7:0] addr,
                    input  wire [7:0] data_w,
                    output wire [7:0] data_r,
                    input  wire       mrq,
                    output reg        mrs,
                    input  wire       re,
                    input  wire       we );

  wire [3:0] w0;
  wire [3:0] w1;
  wire [7:0] w2[0:3];
  wire [3:0] w3[0:7];
  wire [7:0] w4;

  assign w0[0] = ( ~addr[0] & ~addr[1] );
  assign w0[1] = (  addr[0] & ~addr[1] );
  assign w0[2] = ( ~addr[0] &  addr[1] );
  assign w0[3] = (  addr[0] &  addr[1] );

  genvar i;
  genvar j;
  genvar k;
  generate
    for( i = 0; i < 4; i = i + 1 )
    begin:gen_smem_i
      assign w1[i] = we & w0[i];

      for( j = 0; j < 8; j = j + 1 )
      begin:gen_smem_j
        dtype_ff t( .D ( data_w[j] ),
                    .Q (  w2[i][j] ),
                    .en(  w1[i]    ) );

        assign w3[j][i] = w2[i][j] & w0[i];
        assign w4[j]    =          | w3[j];
      end
    end

    for( k = 0; k < 8; k = k + 1 )
    begin:gen_smem_k
      assign data_r[k] = re ? w4[k] : 8'bZ;
    end
  endgenerate

endmodule
```

Listing 8.3 A Verilog module that implements a low-level, 4-element byte-addressable memory.

technologies replace these with more efficient cells for this specific context. The left-hand side of the diagram implements an **address decoder**: the address $addr$ is used to select which row we are dealing with using a similar approach as we have seen when constructing a demultiplexer. For example, if $addr_0 = 1$ and $addr_1 = 0$, we are dealing with row 1; the horizontal **word select** signal for that row is set to 1. In this example, the memory is said to have one **read port** and one **write port** in the sense that one value can be read or one value written at a time. For some applications, it might be advantageous to have more than one read or write port: for

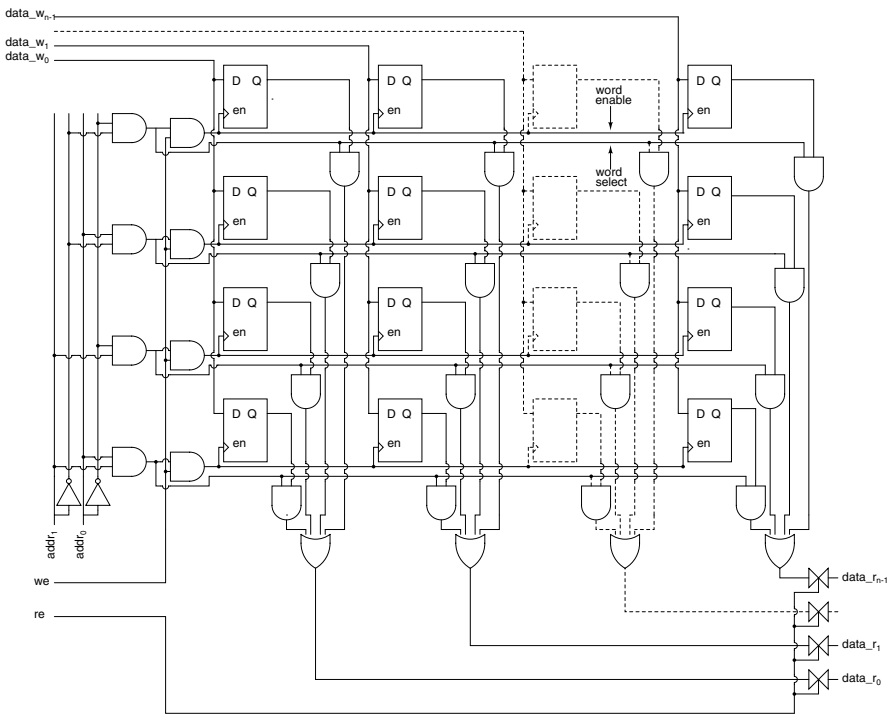


Figure 8.13 A diagram of a low-level, 4-element, *n*-bit memory device.

example, in a memory used as a register file it is common to have two read ports so two operands can be read at the same time. In such a case we can simply replicate portions of the design that are associated with reading or writing and address decoding. The key thing to realise is that this design is very regular; there is a lot of repetition of basic blocks of logic which means fabrication will be somewhat easier. The knock on effect is that the design is also scalable in the sense that we can easily add more elements or ports to the memory if we are willing to pay the associated cost in space.

When we want to read a value from a row, i.e., an address in the memory, we set the **read enable** signal to $re = 1$. Setting the **write enable** signal to $we = 0$ causes the **word enable** signal for the row we are interested in to 0 meaning that the latches do not receive a signal on their *en* inputs and a new value is not stored in the latch. Instead, the Q outputs of each latch in the row are ANDed with the word select signal for that row, so that only the output of that row is selected, and fed through to the memory output $data_r$ to complete the read. Notice that we have fed the output through 3-state enable gates: only when $re = 1$, i.e., we are actually performing a read operation, are the output wires driven with a value. This allows

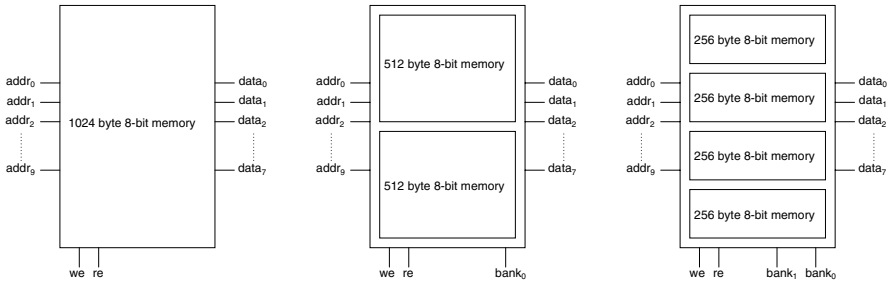


Figure 8.14 Three different organisations for a 1024 byte memory device.

the output wires (or the data bus more generally) to be shared with other devices or perhaps be used for input and output. When we want to write a value in a row we set the read enable $re = 0$ and write enable $we = 1$ which, when combined with the word select for that row, causes a pulse to be sent along the associated word enable signal into the *en* ports for the latches in that row. This causes them all to store the value passed into their *D* input from $data_w$. Notice that there is no handshake logic associated with this implementation: access occurs instantaneously aside from the effects of propagation delay.

The design is implemented in Listing 8.3 using the same interface as previously. The wire vector `w0` and the associated continuous assignments implement the address decoder; one can view components of `w0` as word select signals for the associated row. The rest of the design is built using a generate statement whose main job is to instantiate the D-type flip-flops that actually store values. In addition, components of `w1` act as the word enable signals for the associated row and `w4` collects values from a given column into a single signal that is fed through to the output: we use a final continuous assignment to implement the 3-state style behaviour such that when the `re` signal is low, the memory output drives no value. The associated behaviour when the device is asked to perform the same accesses as the abstract memory is shown in Figure 8.12.

8.1.4 Memory Banking

Implementing a simple memory device as described above is attractive in that increasing the size of memory simply means replicating central parts of the basic design. This is excellent from the point of view of manufacture since replication of similar basic components makes the process much easier. However, if for example you were tasked with building a byte addressable memory device capable of storing 1kB of content, there are several other possible approaches. Instead of designing a single monolithic device which stores 1024 bytes, one might for exam-

ple opt to build the device from smaller memories, say four memories each storing 256 bytes or two memories storing 512 bytes each. This concept is called **banking**, the collection of smaller memories is called a **memory bank**. To allow the user to specify which memory within the bank a particular access is targeting, extra inputs to the memory are required; Figure 8.14 describes three choices for organisation of the 1024-byte memory and adds bank selector signals $bank_0$ and $bank_1$ to address the two and four memories in the respective cases of having a banks of 512 and 256 bytes. The internal logic which uses these signals to access the component memories is not shown; clearly one would expect it to resemble some type of demultipliexer and multiplexer whereby the input and output signals are connected to the component memories depending on the values of the bank selectors.

There are several reasons why this latter approach might be better. Firstly it prevents scale from becoming a problem: although our previous design was easy to add to, a limit emerges where doing so starts to become cumbersome from the point of view of issues such as wire length and so on. Secondly, it allows designers to focus their attention more. That is, one might decide to specialise in designing and building memory devices for use by others in larger systems. By using memory banking the memory designer can concentrate on an efficient design for the basic building block and the system designer can simply use this rather than having to perform any alteration. Introduction of standard memory sizes and interfaces makes this an even more attractive proposition.

Finally, and perhaps most importantly, the use of banked memory allows us to somewhat improve memory access performance. Consider the use of our imaginary 1kB, byte addressed memory within a 32-bit processor design. Since the word size of the processor is 32 bits, loading or storing a word to or from the memory requires four accesses each loading or storing an 8-bit value. With a single monolithic memory, these accesses will probably need to be performed sequentially since the address decoder can only support one access at a time (unless there is more than one read or write port). With the banked design where we use four 256-byte memories, we can split our word into four and store 8-bits of it in each memory in the bank. As a result, since the four 8-bit parts of the word are scattered amongst different memories, they can all be accessed in parallel reducing the time required. This is sometimes called **interleaved access**. To instrument this, we need to feed the bank selector inputs with appropriate values so the right address maps to the right bank. In the case of using four 256-byte memories to build the 1024-byte device, one would drive $bank_0$ and $bank_1$ with address inputs $addr_0$ and $addr_1$ and then disregard these when using them as the actual address fed to the memory (say $addr\prime$). As such, this would result in the following mapping:

```
int main( int argc, char* argv[] )
{
  int A[ 1024 ];
  int B[ 1024 ];
  int C[ 1024 ];

  for( int i = 0; i < 1024; i++ ) {
    A[ i ] = B[ i ] + C[ i ];
  }
}
```

Listing 8.4 An example program to demonstrate the effects of temporal and spatial locality.

$$
\begin{array}{lcll}
addr = 0000_{(2)} & \rightarrow & bank = 00_{(2)} & addr\prime = 00_{(2)} \\
addr = 0001_{(2)} & \rightarrow & bank = 01_{(2)} & addr\prime = 00_{(2)} \\
addr = 0010_{(2)} & \rightarrow & bank = 10_{(2)} & addr\prime = 00_{(2)} \\
addr = 0011_{(2)} & \rightarrow & bank = 11_{(2)} & addr\prime = 00_{(2)} \\
addr = 0100_{(2)} & \rightarrow & bank = 00_{(2)} & addr\prime = 01_{(2)} \\
addr = 0101_{(2)} & \rightarrow & bank = 01_{(2)} & addr\prime = 01_{(2)} \\
addr = 0110_{(2)} & \rightarrow & bank = 10_{(2)} & addr\prime = 01_{(2)} \\
addr = 0111_{(2)} & \rightarrow & bank = 11_{(2)} & addr\prime = 01_{(2)} \\
\vdots & & \vdots & \vdots
\end{array}
$$

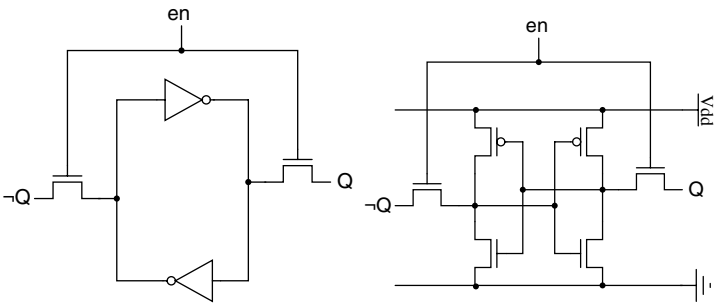
Or, in simpler terms, address 0 maps to address 0 in bank 0 while address 1 maps to address 0 in bank 1 and so on. As a result, we can access all four bytes held in address 0 within banks $0 \ldots 3$ in parallel rather than in sequence.

8.1.5 Access Locality

The principle of **locality** is used to characterise when and where accesses to a storage device are performed. The idea is that given a sequence of accesses, the device will exhibit two different forms of locality:

Definition 34. Temporal locality means that recently accessed locations are likely to be accessed again in the near future. **Spatial locality** means that two locations with addresses that are close to each other will be accessed close together in time.

Listing 8.4 demonstrates both forms of locality. Consider the instructions that make up the function; the loop performs 1024 iterations over instructions that add elements of arrays `B` and `C` together and store the result in `A`. Given one access to the instruction that performs an addition of the *i*-th elements of `B` and `C` it is likely that the instruction will be accessed again. Only when the loop terminates will this be false so 1023 times out of 1024 one access correctly predicts another, i.e., there is temporal locality in the instruction accesses. Now consider the data items in array `A` which we access sequentially. If we access the *i*-th element of `A` there is a good



(a) A functional view. (b) A concrete SRAM cell.

Figure 8.15 A diagram showing transistor level SRAM cell design.

chance we will access the $(i+1)$-th element as well; again this prediction is correct 1023 times out of 1024. Similar arguments can be made for the $(i+2)$-th or $(i+3)$-th elements although access to these given an access to the i-th element becomes increasingly unlikely; either way we clearly have spatial locality in the data accesses.

It turns out that most programs we run on a processor will exhibit both forms of access locality, they are not confined to contrived examples like the above. Thus, given knowledge of the recent past we can, with reasonable accuracy, predict which instructions are likely to be required in the near future. The idea then, is to use the memory hierarchy to improve performance by exploiting the principle of locality. We try to keep often used data and instructions, the **working set**, in the faster parts of the memory hierarchy which are typically smaller and closer to the processor.

8.2 Memory and Storage Specifics

8.2.1 Static RAM (SRAM) and Dynamic RAM (DRAM)

Static Random Access Memory (SRAM) is a method of implementing memory that uses roughly the same technique as using latches or flip-flops; we saw a small-scale example of such an approach earlier on. However, for specific use in our memory design, SRAM cells can be more efficient than a general-purpose latch or flip-flop. Figure 8.15a shows a functional view of an SRAM cell: in concept, it is built from two NOT gates which hold the value in a loop (note that this very roughly matches the approach used by the mercury delay line). The gate behaviour means that one can read the value Q on one side before it is negated so we can read $\neg Q$ on the opposite side. This functional view is implemented in Figure 8.15b using a

total of six transistors, four of which store the 1-bit value and two of which are used to access the value. Notice that the four storage transistors are arranged so that the resulting circuit has two stable states which are used to represent the values 0 and 1 (one can think of these as similar to the stable states in a latch). Access to the cell is performed via a control signal labelled *en*; this signal controls the two access transistors. To read the value stored by the cell, one pre-charges the Q and $\neg Q$ wires to 1. When the *en* signal is set to 1, either $\neg Q$ or Q is discharged depending on the stored value; if 1 is stored then $\neg Q$ is discharged, but if 0 is stored then Q is discharged. To write a value into the cell we discharge Q or $\neg Q$ and pre-charge the other, which then forces the state to be stored as required when the *en* signal is set to 1. Since SRAM cells are based on a similar design to a latch, they only retain their value for as long as they are powered. The cells have a fast access speed, in the order of 10ns, but are quite expensive in terms of the space they require. As a result SRAM is used in situations where performance is more important than size; examples include cache memory and register files.

From a simplistic point of view, **Dynamic Random Access Memory (DRAM)** makes the opposite trade-off to SRAM. Each cell is implemented using a small capacitor that is charged appropriately to store a 1-bit value. Since capacitors leak charge over time, the DRAM cell needs to be periodically refreshed; this dictates some extra, external control logic that implements the refresh cycle at appropriate intervals so that the content is only lost if the device is powered off. Where the refresh logic is built into the memory itself, we sometimes term the result **Pseudo-Static RAM** since it can be used as if it were SRAM even though internally it is DRAM.

Access to the capacitor in a DRAM cell is slow (in the order of 50ns) when compared to an SRAM cell. However, the DRAM cell has a major advantage in terms of size however: it is significantly smaller than an SRAM cell, up to a quarter of the size in fact. This means DRAM cells can be closely packed and result in larger practical memories. Examples of use include main memory which is typically orders of magnitude larger (and slower) than cache and registers. However, there is a large taxonomy of DRAM-based memory designs; all operate in roughly the same way but are typically specialised or improved to suit a particular application:

Video DRAM (VRAM) As described, our normal DRAM architecture has one read and one write port but only one of those can be used at once; one can perform a load or a store but not both simultaneously. This is unattractive in some settings, the obvious example being memory that stores data ready to be sent to some display device. VRAM solves this problem by offering two ports, one for reading data onto the display device and one for the processor to update the data read for display.

Extended Data Out (EDO) DRAM The innovation of EDO DRAM was to use pipelining techniques in respect to memory accesses. That is, an EDO DRAM would allow access to the memory to start before the previous one had entirely completed. Although we discuss pipelining in Chapter 6, this roughly means that the time taken to perform one access (the latency) is unchanged but the number of accesses completed per unit of time (the throughput) is increased.

Synchronous Dynamic RAM (SDRAM) In normal DRAM, an enable signal has to be presented to the memory which provokes it to perform some action; it has to react as quickly as it can when this signal is raised. SDRAM does away with this mode of operation by clocking the memory so that load and store operations can only be initiated on positive clock edges. Thus, the processor is able to control how quickly the memory operates and the added synchronisation allows for more complex pipelining than in simpler designs such as EDO.

Double Data Rate (DDR) SDRAM DDR SDRAM extends basic SDRAM by allowing load and store operations to be initiated on both positive and negative clock edges, thus doubling the rate at which such operations can be performed.

8.2.2 Non-volatile RAM and ROM

Most slower, secondary storage devices such as magnetic and optical disks retain their content even after they are powered off: they are non-volatile. However most faster, primary memories, implemented as SRAM or DRAM for example, only retain their content when powered. If one requires a memory which is non-volatile and yet faster than typical secondary storage devices, some form of specialist device is used. Roughly speaking, non-volatile memories are useful in applications where it is unattractive to use secondary storage devices, due to power or size for example in mobile or embedded computers, or the access latency for a secondary storage device is unacceptably long.

Consider for example the time taken for computer to cold-boot, i.e., load the operating system ready to accept input from the user after being powered on. This delay is a constant source of annoyance; Raskin [57] describes that the Canon CAT computer went to the lengths of saving an image of your screen upon logout and redisplaying it during the boot process to pretend it had completed faster than it had ! Without such trickery, the delay itself is caused largely by the time taken to access the slow secondary storage; if the operating system were stored on a non-volatile primary memory, the (legitimate) boot time could be significantly less.

Of the different approaches to non-volatile memories, major differences exist in how one approaches the construction of ROM and RAM type devices. Although they are potentially much more useful, non-volatile RAM devices are hard to construct. Although there is some recent progress in the design of devices that are close to being fast enough, the typical approach is to couple a conventional low-power RAM with a battery backup which retains the RAM content after the main power source is turned off. Clearly this is not ideal since the battery will eventually run out as well. Even so, this method is used extensively in computers to store critical information: if the clock on your computer suddenly stops working, there is a chance that the battery for this memory has failed ! Devices that cannot be written to, i.e., are read only, are easier to construct; there are roughly four evolutions of device available:

Mask-Programmed ROM The first approach essentially hard-codes the memory content in roughly the same way that a logic circuit would be created. That is, one creates a mask representing the binary content of the memory and uses the photo-lithography process to etch the image onto some substrate, coupled to some logic for accessing it, as described in Chapter 2. Once the memory has been created, the only way to alter it is to build a new one; the process is quite costly and inflexible as a result.

Programmable ROM (PROM) To avoid the problem of having to reconstruct a memory produced by the masking approach, the idea of being able to program a blank device with some image after manufacture was introduced. This process works in a similar way to the PLA device described in Chapter 2 whereby fuses are burnt out or left alone so as to produce the binary content of the memory. Once the image is created, changes are not possible.

Erasable PROM (EPROM) Even PROM type devices are somewhat inflexible in that as well as being read only when being used, they are read only forever. This leads to a slightly different definition of ROM: we are interested in a device that might be read only while in use as a memory, but is writable (or at least erasable to form a blank device) at other times. This is the goal of EPROM memory.

The first EPROM devices, invented by engineer Dov Frohman in 1971, were erasable via exposure to strong ultraviolet light due to their use of special types of transistor. The memory chip was equipped with a transparent window through which a special-purpose machine performed the exposure: the erasure process took about half an hour, the device was then able to retain the memory content for about ten years. It was common for chips to be equipped with a foil cover to stop erasure by sunlight !

Electrically Erasable PROM (EEPROM) The process of erasure in EPROM designs is somewhat cumbersome given the chip needs be taken out of the surrounding circuit and operated on using extra equipment. EEPROM developed on these basic ideas by allowing erasure via an electrical interface and selective erasure so that the memory could be partly updated rather than totally blanked. The trade-off for this is that unlike EPROM that can typically be erased an unlimited number of times most EEPROMs have a limit due to degradation of storage medium, typically of ten thousand or so times.

Flash memory is perhaps the most modern example of the EEPROM; they are fast enough and can be manufactured with enough capacity to replace a read only magnetic disk for example. These types of memory device are commonly used in digital cameras for example.

8.2.3 Magnetic Disks

A magnetic disk is a collection of circular **platters** each of whose surface is coated with a magnetisable material. The idea in terms of storing information on the plat-

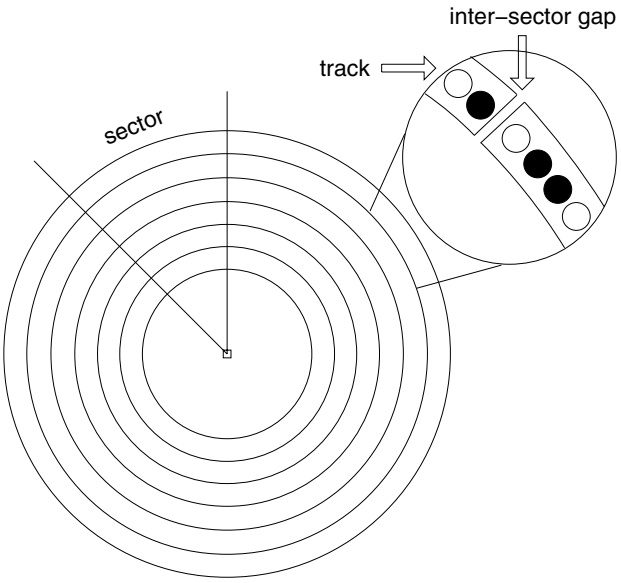


Figure 8.16 A diagram showing tracks and sectors in a typical hard disk format.

ters is that a **disk head** is positioned at a given point above the platter and used to magnetise the surrounding region to encode a single binary digit: the magnetic field is made to align one way to represent 1 and the other way to represent 0. When the information is to be read back, the disk head is positioned in the same place. The magnetic field on the platter induces a current in the disk head which is relayed back as binary data.

To provide some consistency and interoperability between disks and the devices that control them, a standard way of sectioning up the platters is employed. This is the **low-level format** of the disk. Firstly, a disk may consist of a number of platters; each platter has a dedicated disk head. A given platter is divided up into a number of **tracks** which are arranged in concentric circles on the platter surface; tracks in the middle of the platter are shorter than those at the edge. The set of tracks, over all platters, at a given position is called a **cylinder**. Each track is divided up into a number of **sectors** which roughly look like arcs of each concentric circle. Between each sector is a small blank space called the **inter-sector gap**. The end of each sector is punctuated by error correction information that is used to ensure imperfections in the process can be catered for somewhat invisibly for whoever is using the device. Figure 8.16 details these features diagrammatically.

In this context, one can think of some function of platter, track and sector as being the address we are accessing. The access latency is determined by two aspects: the **seek time** which determines how quickly the disk head can be moved to the right

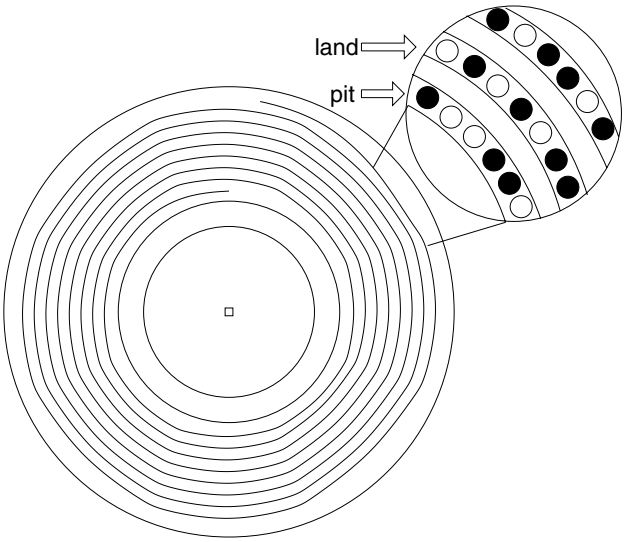


Figure 8.17 A diagram showing tracks and sectors in a typical CDROM format.

track, and the **rotational latency** which determines how quickly the right sector arrives under the disk head as the disk spins. With so-called hard disks the disk head floats above the platter on a cushion of air and can move very quickly, with floppy disks the disk head is actually in contact with the platter. This can cause the disk head and platter to degrade over time; the reliability of floppy disks is questionable as a result.

8.2.4 Optical Disks

Optical disks take a somewhat similar approach to magnetic disks in the sense that information is encoded on the surface of a (single) platter. Significant differences exist, however, in the way the information is encoded and organised. Instead of magnetic regions information is encoded, as binary digits, using a series of physical indentations in the surface; the indentations are termed **pits**, the untouched regions between pits are termed **lands**. These features are used to encode 1 and 0, although not directly but as part of some more complicated scheme (i.e., a pit or land does not directly encode a 1 or 0). Information is read by positioning a laser, the **read head**, over the relevant region and firing it at the platter. The nature of the surface is such that although both pit and land will reflect the laser, they are such that the reflection has its phase shifted depending on what it is fired at: this change in phase

is detected by a device that monitors the reflected beam and is converted into binary data.

The information is encoded as a series of **sectors** divided into **frames**; the information forms a single spiral track on the platter. Header and error correction information are appended to each frame. Reading the frames is complicated by the fact that the spiral is tighter at the middle of the platter than at the edge. Using a **Constant Linear Velocity (CLV)** scheme means the platter speed is variable but the pits pass at a constant speed: however, the drive mechanism needs to be quite complex as a result. In contrast, using a **Constant Angular Velocity (CAV)** scheme means the platter speed is constant but the pits pass at a variable rate which leads to faster transfer when reading from the edge than the middle.

Common forms of optical disks include the **Compact Disc Read Only Memory (CDROM)** and **Digital Versatile Disc (DVD)** both of which are read-only devices; their content is created using a one-off process called **mastering** whereby the pits are permanently etched on the platter. The so-called **Rainbow Book** standards dictate how information in the frames is interpreted: **Red Book** specifies the format for audio CDs, **White Book** specifies the format of **Video CDs (VCDs)**, and **Yellow Book** specifies the general CDROM format. Read-write versions, for example **Compact Disc Read-Write (CDRW)**, are possible using a coating on the platter. The coating can be selectively heated using an infra-red laser to crystallise or anneal the platter and create or remove pits. This process can repeated around a thousand times before the platter surface degrades and the encoding is rendered permanent.

8.2.5 Error Correction

Digital components sometimes fail; unlike failures in a processor which would typically result in catastrophic failure, failures in memory or a storage device can be somewhat more subtle. For example, consider if a single memory cell were to fail: this would produce errors so that values transmitted from memory to processor would be erroneous in just one bit. This might not cause a program to crash, it might just produce slightly odd results. In addition, both memory and storage devices are typically implemented as separate devices to the processor so errors might also exist in transmission rather than storage of values. We can categorise either type of error as either **permanent** or **transient**. Permanent errors are always present, they behave in the same way at all times. Transient errors only occur occasionally so that, for example, a read from memory might be completed without error at one point in time but produce an erroneous value at another. Sometimes these are termed **hard errors** and **soft errors** respectively.

Techniques for **error detection** and **error correction** partly accommodate the inevitable occurrence of errors in storage and communication devices. In our discussion we will refer to transmission of values; it is important to see that this is as relevant to retrieval of values from a storage device by the processor as it is to communication of values along some medium. Error detection hopes to identify when

an error has occurred. This can either result in the processor being signalled so that for example it can repeat an access in hope of a subsequent success, or simply to highlight erroneous memory so that it can be replaced. Error correction takes this one step further so that errors can be corrected and execution can continue without further intervention; this helps to accommodate the erroneous memory without need for immediate replacement. Roughly speaking both methods add redundant information to the value held in memory so that errors in the value can be detected and potentially corrected; the more redundant information that is added, the more errors can be detected or corrected.

8.2.5.1 Parity Codes

Probably the most basic form of error detection is the **parity code**. The basic idea is that one should count the number of bits set to 1 in x, that is, the Hamming weight of x, and call this total t. We then append a bit p to x based on the value of t to get a new value $x' = x : p$ which we call the **code word**. To use **even parity** the bit p should be 1 if t is odd, thus making the total number of bits set to 1 in the code word x' even. To use **odd parity** the reverse is true; the bit p should be 1 if t is even, thus making the total number of bits set to 1 in x' odd. As an example, consider the value $x = 1001_{(2)} = 9_{(10)}$, since there are two bits set to 1 in x, we compute x' using even and odd parity as

$$\begin{aligned} \text{EVEN-PARITY}(x) &= 10010 \\ \text{ODD-PARITY}(x) &= 10011 \end{aligned}$$

with the parity bit appended in the least-significant position in this case.

Regardless of whether even or odd parity was used, the code word x' can now be checked for errors using the extra bit. Specifically, if an odd number of errors occur in transmitting x' then we can detect this because the parity bit will differ from that expected. For example, imagine we construct the even parity-based encoding of $x = 1001_{(2)}$ as $x' = 10010$. Now imagine that instead of transmitting the code word $x' = 10010$ there is some error which flips one bit and causes us to transmit $y' = 11010$ instead. After extracting the parity bit $p' = 0$ we can count the number of bits set to 1 in y'; this time there are three. Since we are using an even parity scheme we would expect the parity bit to be 1 so that the overall number of bits set to 1 is even. But $p' = 0$ so there must have been some error.

The basic parity code is easy and inexpensive to implement; it just requires a few extra XOR gates to compute and check the parity bit. However, there are two main problems. Firstly we cannot tell where the error occurred or how to correct it, the value must be resent and we must hope that the error is transient; secondly if an even number of bits are flipped, then a single parity bit can no longer detect the error. One can address the second problem by using a more involved scheme such as the **Cyclic Redundancy Check (CRC)**; the first problem requires some more ingenuity.

8.2.5.2 Hamming Codes

The **Hamming Code**, named after their inventor Richard Hamming, improves on simple error detection mechanisms by allowing the correction of errors once they are detected. This required that more than one bit be appended to the value; Hamming used the notation (n,m)-code to denote a scheme where there were n bits in the code word of which m where actually the original value and $n-m$ were added afterwards to instrument error detection and correction.

The general Hamming Code is somewhat involved to describe, so we concentrate on the specific and common example of a $(7,4)$-code introduced in 1950. The basic idea is to have several parity bits, in this case three. In a code word under this scheme, all bits in the code word whose index i implies $i+1$ is a power-of-two are parity bits; this means bits x'_0, x'_1 and x'_3, denoted p_0, p_1 and p_3, are parity bits since $0+1=2^0$, $1+1=2^1$ and $3+1=2^2$. The four other bits represent the original value. Each parity bit deals with a sub-set of the code word: p_0 deals with the parity of x'_2,x'_4 and x'_6; p_1 deals with the parity of x'_2,x'_5 and x'_6; p_4 deals with the parity of x'_4,x'_5 and x'_6. Adopting a vector notation, our code word is thus

$$x' = (p_0, p_1, x_0, p_3, x_1, x_2, x_3)$$

where x_i are from the original value and p_j are our parity bits as described above and calculated as

$$\begin{aligned} p_0 &= x'_2 \oplus x'_4 \oplus x'_6 \\ &= x_0 \oplus x_1 \oplus x_3 \\ p_1 &= x'_2 \oplus x'_5 \oplus x'_6 \\ &= x_0 \oplus x_2 \oplus x_3 \\ p_3 &= x'_4 \oplus x'_5 \oplus x'_6 \\ &= x_1 \oplus x_2 \oplus x_3. \end{aligned}$$

Encoding the value $x = 1001_{(2)} = 9_{(10)}$ using even parity would result in the bits $p_0 = 0$, $p_1 = 0$ and $p_3 = 1$ so that the code would be computed as $x' = 1001100$. An error in any one bit of the original value embedded in this code would affect the parity bits; in fact, which bits they affect will unique in the sense that by checking all three we can determine the location of the error. Consider flipping x_0 which corresponds to x'_2 so that instead of transmitting the code word $x' = 1001100$ we transmit $y' = 1001000$. Recalculating the parity bits we find that

$$\begin{aligned} p'_0 &= y'_2 \oplus y'_4 \oplus y'_6 \\ &= 1 \\ p'_1 &= y'_2 \oplus y'_5 \oplus y'_6 \\ &= 1 \\ p'_3 &= y'_4 \oplus y'_5 \oplus y'_6 \\ &= 1. \end{aligned}$$

Since $p'_3 = p_3$ but $p'_0 \neq p_0$ and $p'_1 \neq p_1$ there is an error. To find where the error is, we form a vector from the recalculated parity bits

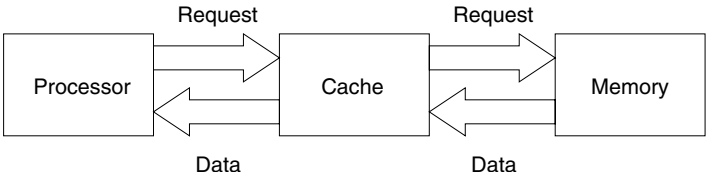


Figure 8.18 A basic block diagram of cache operation.

$$k = (p'_0 \neq p_0, p'_1 \neq p_1, p'_3 \neq p_3)$$

and consider this as an integer; the error is then at bit $k-1$. That is,

$$k = (1,1,0)$$

so we have $k = 011_{(2)} = 3_{(10)}$ and the error is detected in bit $k-1 = 2$. This error can now be corrected by flipping y'_2 to get the original, error-free code word x' back again. Consider another example, this time flipping bit x_3 which corresponds to x'_6 so that we transmit $y' = 0001100$. We compute

$$\begin{aligned} p'_0 &= y'_2 \oplus y'_4 \oplus y'_6 \\ &= 1 \\ p'_1 &= y'_2 \oplus y'_5 \oplus y'_6 \\ &= 1 \\ p'_3 &= y'_4 \oplus y'_5 \oplus y'_6 \\ &= 0 \end{aligned}$$

and find that $p'_0 \neq p_0$, $p'_1 \neq p_1$ and $p'_3 = p_3$. Writing the vector of parity bits out we find that $k = 111$ so that the error is in bit $k-1 = 6$. This can now be corrected by flipping bit y'_6 to get back to compute the correct code word x'.

One can show that this $(7,4)$-code can correct 1-bit errors, or detect but not correct 2-bit errors. Although the length of the code word is larger we do not need to retransmit very often because unless there are very many errors, they can be fixed automatically.

8.3 Basic Cache Memories

A **cache** is a small area of fast RAM and associated control logic which is placed between the processor and main memory as shown in Figure 8.18. The cache is typically invisible to the programmer; the interface between processor and main memory is unchanged and from a functional point of view, and accesses are serviced as if the cache was not there. The basic idea is that since the cache is smaller

than main memory, it stores a sub-set of the memory content. However, since the cache can be implemented using a faster technology than main memory, accesses to content in the cache are faster than those which are not. As a result of locality in the incoming address stream, the cache reduces the load on the rest of the memory hierarchy because it can be managed to hold the current working set of data and instructions.

The cache is typically organised as a number of **cache lines**, each of which comprises a number of **sub-words** that are used to store contiguous addresses from main memory:

Definition 35. A cache C is constructed from C_{lines} cache lines. The i-th cache line, which we write as $C[i]$, has three fields:

- $C[i]_{tag}$ is a **tag** used as an identifier for the content within the cache line.
- $C[i]_{valid}$ indicates whether the content is valid or not (i.e., if there is content in the cache line).
- $C[i]_{data}$ is the actual cache line content which consists of C_{words} sub-words each of whose size matches the resolution of the memory. Where appropriate we refer to the j-th sub-word within the i-th cache line as $C[i]_{data[j]}$.

To make life easier, we constrain C_{lines} and C_{words} so they can only take values which are powers-of-two; given a total cache size of 2^n sub-words for some n and a line size of $C_{words} = 2^m$ for some m, the number of lines is $C_{lines} = 2^{n-m}$. For byte addressed memory and a total cache size of $2^{10} = 1024$ bytes so that $n = 10$ for example, there are many different ways to organise things. We might select any one of the following three options:

$$\begin{aligned} C_{words} &= 2^2 \\ &= 4 \\ C_{lines} &= 2^{10-2} \\ &= 256 \end{aligned} \qquad \begin{aligned} C_{words} &= 2^5 \\ &= 32 \\ C_{lines} &= 2^{10-5} \\ &= 32 \end{aligned} \qquad \begin{aligned} C_{words} &= 2^8 \\ &= 256 \\ C_{lines} &= 2^{10-8} \\ &= 4 \end{aligned}$$

Accesses that are serviced by the cache are termed **cache-hits** and are completed very quickly; accesses that are not held by the cache are termed **cache-misses** and take much longer to complete since main memory must be accessed. When the cache accesses main memory, it is said to have **fetched** data: typically an entire line is fetched in one go. Selecting the best cache organisation is therefore a trade-off: more lines typically mean we can better exploit temporal locality and are quicker to fetch, longer lines mean we can better exploit spatial locality but take more time to fetch. Standard literature categorises cache misses into three classes [29]:

Definition 36. Compulsory misses are unavoidable misses caused by the first access to an element in memory. **Capacity misses** are misses that are independent of all factors aside from the size of the cache. **Conflict misses** are misses where one element replaces another in the cache.

Since the principle of locality guarantees we should get more cache-hits than cache-misses, performance of the average case program is improved. Common measures

of cache effectiveness are the **hit ratio** and **miss ratio** which are calculated as

$$\begin{aligned}\text{HIT-RATIO} &= \frac{\text{TOTAL-HITS}}{\text{TOTAL-ACCESSES}} \\ \text{MISS-RATIO} &= \frac{\text{TOTAL-MISSES}}{\text{TOTAL-ACCESSES}} \\ &= 1 - \text{HIT-RATIO}\end{aligned}$$

It is quite important to stress that performance of only average case programs is improved. That is, if we have a program that behaves somewhat oddly, for example it accesses data in a manner which does not respect the principle of locality, the cache might not be effectual at all.

Beyond the hit and miss ratios, an important performance metric is the idea of **mean access time**, the average length of time for a memory access to complete. If T_{cache} and T_{memory} denote the time required to perform an access to the cache and memory respectively, we calculate mean access time as

$$T_{cache} + T_{memory} \cdot \text{MISS-RATIO}$$

This is easy to see: to access the memory we first check in the cache and if the value is not present, we need to access the memory. So we pay the price of accessing the cache plus the price of access to memory for those proportion of accesses that provoke cache misses. As the miss ratio tends to 0, mean access time tends to T_{cache}. As the miss ratio tends to 1, mean access time tends to $T_{cache} + T_{memory}$. The principle of locality should ensure the miss ratio is close to 0; if it does not then the addition of a cache can actually slow down the system ! To get any meaningful speed-up, we need that

$$\text{HIT-RATIO} > \frac{T_{cache}}{T_{memory}}.$$

To design a real cache using all this theory, we need to consider several key issues:

Addressing Policy Given an access by the processor to an address in memory, how do we decide where this address should map to in the cache given that it only stores a sub-set of the memory content ? Application of this mapping is usually termed **address translation**.

Fetch Policy Given an access by the processor to an address in memory such that the location is not resident in the cache, how and where does one load data from main memory into the cache to satisfy the request ? Furthermore, how does one cope with the fact that data may already be resident in cache location we want to fetch new data into ?

Write Policy Given an access by the processor to an address in memory, how does the cache interact with the memory so that the processor's view of memory is consistent when the access is a store ?

In the next sections we explain different fetch and write policies that are common between most cache architectures. We then introduce three different cache archi-

tectures that use different addressing policies to implement different performance trade-offs. Although there can be some differences (e.g., we typically only load instructions rather than load and store them), since data and instructions are essentially the same data and instruction caches are largely the same as well: to make things easier we concentrate only on data caches.

8.3.1 Fetch Policy

When the processor tries to access an address in memory through the cache and the cache does not contain the corresponding data, it is **fetched** from memory into the cache; once it is located in a cache line it is said to be **resident** in the cache. In order to take advantage of the relative speeds and transfer sizes, the cache typically fetches an entire cache line at once. That is, if the processor tries to access address x, then the addresses around x that map into the same line will be fetched into the cache as well. The idea here is to take advantage of any locality: if x has been accessed then it is probable that the addresses close to x will be accessed soon; if they are, then the line will already be resident and a cache hit will occur rather than the cache miss that resulted from the access to x.

The address x is mapped to a cache line by the address translation mechanism. Once we know which line to fetch the corresponding data into, we need to check if there is already data resident in that line. Remember that the cache only holds a sub-set of the memory content so the resident data need not be that relating to x: if it were, we would have found it, signalled a cache hit and not needed to fetch the line for x at all ! So if there is already data resident, we need to **evict** it from the cache and replace it with the data we do want. If the data has not been altered by a write, this process is simply a case of overwriting what is already there. If it has been altered, we have a number of options which are discussed later when examining write policy.

8.3.1.1 Fetch Ordering

Consider the following example situation where we have a 1024-byte cache with 256 lines and 4 sub-words per-line. The processor issues the cache a load request for address 2. Suppose the address translation maps address 2 onto cache line 0. The line will hold data for addresses $0,1,2$ and 3 but does not currently hold this data, we need to fetch it; the cache needs to load data at addresses $0,1,2$ and 3 in main memory to fill line 0. The question is which order should we load these addresses; there are two main choices:

Critical Word First In this scheme we would load the words in the order $2,3,0,1$. That is, the address we are looking for is loaded first, then the others in cyclic order.

Natural Word First In this scheme we would load the words in the order $0, 1, 2, 3$. That is, the first address in the cache line is loaded first followed by all the others.

Clearly each has some advantages and disadvantages to both schemes. Loading the critical word first means the cache can complete the request from the processor faster since it does not need to wait for other loads from main memory. That is, once it loads the value at address 2 it can return this to the processor and carry on with the others behind the scenes; the processor can hence continue execution earlier. On the other hand, this requires a more complex operational mechanism; loading the natural word first is much easier and leads to a simpler design.

8.3.1.2 Pre-fetching

We have already seen that a cache-hit can only occur if the corresponding data is already resident in the cache; if it is not, a cache-miss occurs and we have to load it from main memory. One method of trying to reduce the number of cache-misses is, given previous a list of previous accesses, to guess which address will be accessed next and fetch this into the cache speculatively. If the guess is right, then a cache-hit occurs and execution continues without the penalty associated with a cache-miss. If the guess is wrong, we do no worse than the cache-miss that would have occurred anyway. This technique is called pre-fetching and is typically realised by a second hardware device that works alongside the cache.

This seems an ideal solution and, if we had a good enough guessing strategy, it is indeed ideal in theory. In practise, such a guessing strategy is hard to construct for general programs even given their access locality patterns. Furthermore, aggressive pre-fetching introduces the new problem of **cache pollution**. For example, consider the case where we pre-fetch some data based on our guess and this fetch evicts data that would have been useful in the near future. We are essentially polluting the cache with useless data. Pre-fetch systems need to carefully balance their operation as a result; generally it is only a good idea to employ pre-fetch if the access pattern is very regular and hence easy to predict with high accuracy.

8.3.2 Write Policy

So far we have ignored what happens when the processor wants to store data into memory: the processor issues the store operation as usual, but what should the cache do ? There are two main problems we need to tackle. Firstly, if we update the data in the cache, what happens to the data in memory ? Secondly, what happens if the data we want to update is not even in the cache at all ? No matter what solution is employed the *key* requirement is that we maintain **consistency** between what is stored in the cache and what is stored in memory. That is, it should be impossible to get confused about the value held in a particular memory location.

There are two main options as regards solving the problem of maintaining consistency between the cache and memory content:

Write-Through When a store is issued by the processor, a write-through cache updates the data in the cache, potentially fetching data into the cache if it is not already present, but also sends the write through to the main memory. This is the simplest choice of policy: since the cache and memory content is always the same, it is always consistent. However, one needs to be careful that the cache still improves performance by eliminating unnecessary accesses to main memory. For example, if we perform a number of writes to the same address, there is no need to keep updating the memory.

Write-Back When a store is issued by the processor, a write-back cache updates the data in the cache but does not immediately write the data through to main memory. The hardware for a write-back cache is more complex, it has to remember which lines are inconsistent with main memory; these are said to be **dirty**. The cache needs to include an additional field for each line, say $C[i]_{dirty}$, which determines whether the content is dirty or not. When a line is evicted, the associated dirty flag is checked to see if it should be written to main memory. This scheme removes unnecessary memory traffic since now we only write to main memory when we really have to.

When the processor performs a load operation and the corresponding cache line is not resident, it is read into the cache. When a store operation is executed, this implication is not necessarily true. If the cache line is resident the store can update local data in the cache; however, if the cache line is not resident we cannot do this: to preserve consistency we need to take into account that there will be other memory locations in the same line. For example, say an example cache has four sub-words per-line and addresses $0, 1, 2$ and 3 map to line 0. If a store is executed using address 2 and the line is not resident, we cannot just update the cache line because what values would the sub-words for addresses $0, 1$ and 3 hold: the cache and memory would be inconsistent. Hence we have two main choices:

Allocate-on-Write Follow the implication of a read, that the data will be accessed again in the near future, and fetch the cache line into the cache before completing the write operation on the resident line.

No Allocate-on-Write Avoid fetching the cache line into the cache and pass the store directly onto main memory; thus cache lines are only fetched into the cache by load operations.

8.3.3 Direct-Mapped Caches

To construct a cache, we need an address translation mechanism that takes an incoming address that results from a load or store operation, and maps it onto the specific sub-word within a cache line which holds the associated data. A **direct-mapped cache** takes a simple approach to this by operating such that each 2^n-th

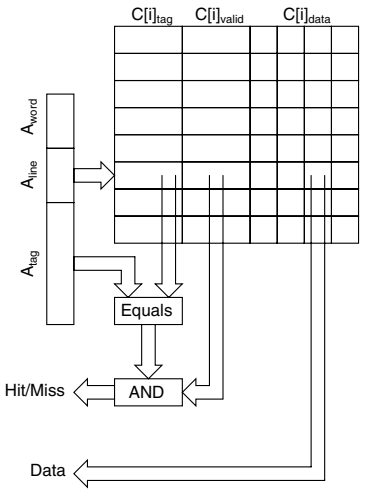


Figure 8.19 A block diagram of a direct-mapped cache.

address in memory maps to the same sub-word in the cache. The address translation mechanism calculates two components which are used to access the cache content:

$$\begin{aligned} A_{word} &= A \bmod C_{words} \\ A_{line} &= \lfloor A/C_{words} \rfloor \bmod C_{lines} \end{aligned}$$

The value A_{line} identifies the cache line for address A, while the value A_{word} identifies the sub-word within that cache line. Since both C_{lines} and C_{words} are selected to be powers-of-two, calculating these components is not costly. Given a 32-bit address and taking $C_{lines} = 2^3 = 8$ and $C_{words} = 2^2 = 4$ so the total cache size is 32 bytes, this amounts to taking the least-significant two bits of A as A_{word} and the next least-significant three bits as A_{line}. For example, given the address

$$A = 00000000000000000000000001\underbrace{000}_{A_{line}}\underbrace{10}_{A_{word}}{}_{(2)} = 34_{(10)}$$

we calculate $A_{line} = 000_{(2)} = 0_{(10)}$ and $A_{word} = 10_{(2)} = 2_{(10)}$.

However, since a given cache can only hold a sub-set of memory content, it follows that there is a many-to-one mapping of memory addresses to cache sub-words. For example, setting $A = 34_{(10)}$ or $A = 66_{(10)}$ gives the same values for A_{word} and A_{line} in both cases: we need a way to distinguish the cases from each other. To achieve this, each cache line holds a tag; we need to calculate

$$A_{tag} = \lfloor \lfloor A/C_{words} \rfloor / C_{lines} \rfloor$$

	$C[i]tag$	$C[i]valid$	$C[i]data[0]$	$C[i]data[1]$	$C[i]data[2]$	$C[i]data[3]$
$i=0$		0				
$i=1$		0				
$i=2$		0				
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

(a) Cache is initially empty.

	$C[i]tag$	$C[i]valid$	$C[i]data[0]$	$C[i]data[1]$	$C[i]data[2]$	$C[i]data[3]$
$i=0$	0	1	0	1	2	3
$i=1$		0				
$i=2$		0				
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

(b) Access to address 1 misses, line 0 is filled.

	$C[i]tag$	$C[i]valid$	$C[i]data[0]$	$C[i]data[1]$	$C[i]data[2]$	$C[i]data[3]$
$i=0$	1	1	32	33	34	35
$i=1$		0				
$i=2$		0				
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

(c) Access to address 34 misses, line 0 evicted then re-filled; access to address 35 then hits in line 0.

	$C[i]tag$	$C[i]valid$	$C[i]data[0]$	$C[i]data[1]$	$C[i]data[2]$	$C[i]data[3]$
$i=0$	1	1	32	33	34	35
$i=1$	1	1	36	37	38	39
$i=2$		0				
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

(d) Access to address 36 misses, line 1 is filled; access to address 37 then hits in line 1.

	$C[i]tag$	$C[i]valid$	$C[i]data[0]$	$C[i]data[1]$	$C[i]data[2]$	$C[i]data[3]$
$i=0$	0	1	0	1	2	3
$i=1$	1	1	36	37	38	39
$i=2$		0				
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

(e) Access to address 1 misses, line 0 evicted then re-filled; access to addresses 38 and 39 then hit in line 1.

	$C[i]tag$	$C[i]valid$	$C[i]data[0]$	$C[i]data[1]$	$C[i]data[2]$	$C[i]data[3]$
$i=0$	0	1	0	1	2	3
$i=1$	1	1	36	37	38	39
$i=2$	1	1	40	41	42	43
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.

(f) Access to address 40 misses, line 2 is filled; access to address 41 then hits in line 2.

Figure 8.20 An example of direct-mapped cache operation.

to check that the cache line content is actually what we expected. Again, since C_{lines} and C_{words} are selected to be powers-of-two, calculating A_{tag} amounts to simply taking the remaining twenty seven bits of the 32-bit address, i.e.,

$$A = \underbrace{00000000000000000000000001}_{A_{tag}}\underbrace{000}_{A_{line}}\underbrace{10}_{A_{word}}{}_{(2)} = 34_{(10)}.$$

Now we find that setting $A = 34_{(10)}$ results in

$$A_{tag} = 00000000000000000000000001_{(2)} = 1_{(10)},$$

but if we instead set $A = 66_{(10)}$, then

$$A_{tag} = 00000000000000000000000010_{(2)} = 2_{(10)},$$

i.e., we can tell when the cache line contains content relating to one address versus the other. Note that in theory we could use the entire address A as the tag since this would obviously permit distinguishability. However, this is wasteful: the least-significant bits are implicit, or the same, for a given line so by not storing this portion we reduce the amount of logic required to implement the cache.

Figure 8.19 shows a block diagram of a typical direct-mapped cache design. The basic idea is that given an address A, the cache calculates A_{word}, A_{line} and A_{tag}. It then inspects the value

$$C[A_{line}]_{valid}.$$

If the cache line does not contain valid content, the sub-word we want is not resident and a cache-miss occurs. However, if the cache line does contain valid content, the cache inspects the value

$$C[A_{line}]_{tag}$$

and compares this to A_{tag}. If the tags do not match, the sub-word we want is not resident and a cache-miss occurs. However, if the tags do match, then a cache-hit occurs and the sub-word we want is resident in

$$C[A_{line}]_{data[A_{word}]}.$$

Imagine we continue to consider the example cache from above, and feed it a stream of memory accesses whose addresses are

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

With our address translation scheme, this produces the cache accesses

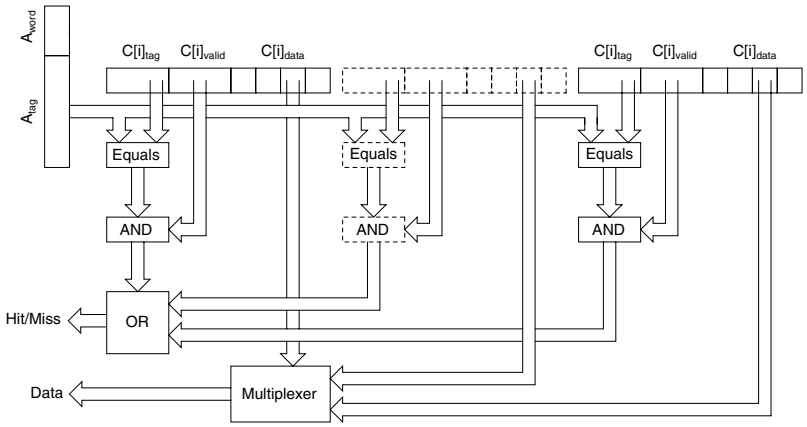


Figure 8.21 A block diagram of a fully-associative cache.

$$
\begin{array}{llll}
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 38 & A_{word} = 2, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{hit .}
\end{array}
$$

The sequence produces five cache hits and five cache misses; Figure 8.20 details the sequence graphically. The first access to address 1 is a compulsory cache-miss since this is the first time we have used this address; the line for address 1 is then fetched into the cache and the correct value at offset 1 is returned as the result. The second access to address 34 maps to the same line, i.e., line 0, as the address 1 whose corresponding line is already resident. This is a conflict cache-miss so the line for address 1 is evicted and replaced by the line for address 34 that is fetched from memory; notice that these are distinguishable since they have different tags. The third access to address 35 provokes a cache-hit since the corresponding line, fetched when we accessed address 34, is resident.

8.3.4 Fully-Associative Caches

A direct-mapped cache can only ever have one choice of line per address; this can lead to what is called **cache interference** or **cache contention**. This was evident in a previous example where addresses 1 and 34 interfered when they were both mapped to line 0. When the resulting eviction happens repeatedly, we say that the access stream thrashes the cache, a situation where performance is significantly degraded.

One approach to solving this problem is to use the idea of **associativity**. An associative cache allows any memory location to map into one of many cache lines; a **fully-associative cache** allows a memory location to map to any cache line. Since the required data could be anywhere in the cache, searching for it is more complex and the hardware realisation is typically larger and more power hungry. However, associative caches can exhibit a lower miss ratio than a direct-mapped cache. Again considering the previous example, if addresses 1 and 34 were mapped to different lines rather than the same line, then the number of cache-hits would be increased.

Since any memory address can map to any cache line, the problem remains how to construct an address translation scheme. Firstly we consider how to actually decide which line, i.e., the value A_{line} to use when we fetch data into the cache. Given searching for an empty line is typically a bad idea since it is just too costly, we have a number of options. For example, we might select one of the following:

Random Using a pseudo-random number generator, a line number is chosen at random regardless of the address and what is already resident in the cache. This is a low-cost option but can suffer from problems in that "unlucky" random sequences still provoke cache contention; the hope is that on average the randomisation will give a good distribution of line usage and hence low levels of contention.

Least Recently Used (LRU) Each time a cache line is accessed, it is moved to the head of a recently used (LRU) list. When the cache is required to load new data, the line at the end of the recently used list is selected; the rationale is that this is the least recently used so is most probable never to be used again. Normally, the cache will be quite large so maintaining the LRU list is usually too costly an option.

The word offset A_{word} is simply given by the least-significant m bits of the address A, i.e., $A_{word} = A \bmod C_{words}$, in both cases; the tag A_{tag} is given by the remaining bits.

Now, given we have translated the address to a line when fetching the data we need to consider how to find it again when we want to use it. Essentially, since a fully-associative cache allows a memory address to map into any cache line we need to search the entire cache to see if it is resident. Since cache access needs to be as fast as possible, a linear (or even binary) search is out of the question; we must search all cache lines in parallel. This feature contributes massively to the cost of a hardware realisation; Figure 8.21 details a block diagram of the design.

Again consider an example fully-associative cache for a byte-addressed memory where we have that $C_{lines} = 2^3 = 8$ and $C_{words} = 2^2 = 4$ so the total cache size is 32

bytes. Imagine we feed it the same stream of memory accesses as before:

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

It is not difficult to see that using a random line selector, we can achieve optimal performance. For example, if our pseudo-random number generator happens, very fortuitously, to give the sequence

$$0, 1, 2, 3, 4, \ldots$$

then our access pattern is

$$\begin{array}{llll}
A = 1 & A_{word} = 1, A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 34 & A_{word} = 2, A_{tag} = 8 & A_{line} = 1 & \rightarrow \text{ miss} \\
A = 35 & A_{word} = 3, A_{tag} = 8 & A_{line} = 1 & \rightarrow \text{ hit} \\
A = 36 & A_{word} = 0, A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ miss} \\
A = 37 & A_{word} = 1, A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ hit} \\
A = 1 & A_{word} = 1, A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 38 & A_{word} = 2, A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ hit} \\
A = 39 & A_{word} = 3, A_{tag} = 9 & A_{line} = 2 & \rightarrow \text{ hit} \\
A = 40 & A_{word} = 0, A_{tag} = 10 & A_{line} = 4 & \rightarrow \text{ miss} \\
A = 41 & A_{word} = 1, A_{tag} = 10 & A_{line} = 4 & \rightarrow \text{ hit .}
\end{array}$$

The only misses that remain are compulsory. However, the pseudo-random number generator is equally likely, but rather less fortuitously, to produce the sequence

$$0, 0, 0, 0, 0, \ldots$$

in which case the cache performs much worse:

$$\begin{array}{llll}
A = 1 & A_{word} = 1, A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 34 & A_{word} = 2, A_{tag} = 8 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 35 & A_{word} = 3, A_{tag} = 8 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 36 & A_{word} = 0, A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 37 & A_{word} = 1, A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 1 & A_{word} = 1, A_{tag} = 0 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 38 & A_{word} = 2, A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 39 & A_{word} = 3, A_{tag} = 9 & A_{line} = 0 & \rightarrow \text{ hit} \\
A = 40 & A_{word} = 0, A_{tag} = 10 & A_{line} = 0 & \rightarrow \text{ miss} \\
A = 41 & A_{word} = 1, A_{tag} = 10 & A_{line} = 0 & \rightarrow \text{ hit .}
\end{array}$$

For the average case we are neither especially lucky nor unlucky, and as a result can typically expect better hit ratios than a direct-mapped cache.

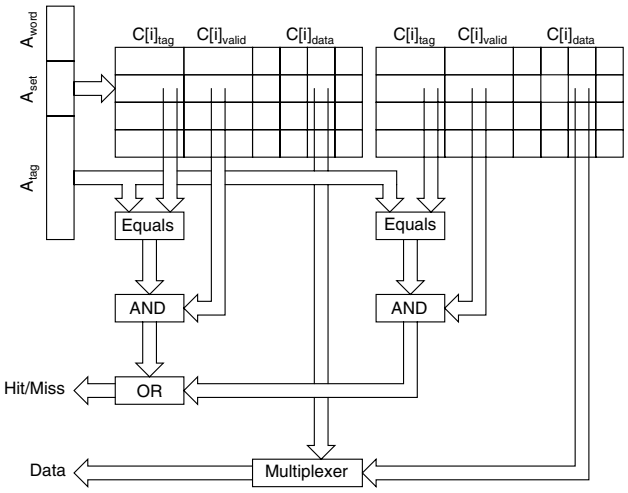


Figure 8.22 A block diagram of a set-associative cache.

8.3.5 Set-Associative Caches

On one hand, direct-mapped caches are simple to build but *can* exhibit relatively poor performance. On the other hand, fully-associative caches *can* improve performance but are more costly to realise. What would be ideal is to find a trade-off between the two designs; the **set-associative cache** does just this.

The basic idea is that the total cache size is sectioned into C_{set} **sets**, each of which groups $N = C_{lines}/C_{set}$ cache lines together. Instead of mapping an address to a cache line we instead map to a set, within each set, searching for a particular line is performed in a fully-associative manner. Usually we say such a cache is N-way set-associative since there are N places a cache line can reside in each set. Figure 8.22 gives a rough block diagram of set-associative cache design where $C_{set} = 2$. In a similar way to before, we define our address translation scheme as

$$\begin{aligned} A_{word} &= A \bmod C_{words} \\ A_{set} &= \lfloor A/C_{words} \rfloor \bmod C_{set} \\ A_{tag} &= \lfloor \lfloor A/C_{words} \rfloor / C_{set} \rfloor \end{aligned}$$

so that the word offset is taken as the bottom m bits of the address A and the set number A_{set} is the next $\lceil \log_2(C_{set}) \rceil$ bits; since we typically constrain C_{set} to be a power-of-two this value is an integer and the address translation scheme is easy to implement efficiently. To decide which line within the set is used, we just select a policy that follows our original choices for the fully-associative cache. For example, we might select a random or LRU selection policy. Since C_{set} is usually small (say

	Set 0						Set 1					
	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$
$i=0$		0						0				
$i=1$		0						0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(a) Cache is initially empty.

	Set 0						Set 1					
	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$
$i=0$	0	1	0	1	2	3		0				
$i=1$		0						0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(b) Access to address 1 misses, line 0 in set 0 is filled.

	Set 0						Set 1					
	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$
$i=0$	0	1	0	1	2	3		0				
$i=1$	4	1	32	33	34	35		0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(c) Access to address 34 misses, line 1 in set 0 is filled; access to address 35 then hits in line 1 of set 0.

	Set 0						Set 1					
	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$
$i=0$	0	1	0	1	2	3	4	1	36	37	38	39
$i=1$	4	1	32	33	34	35		0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(d) Access to address 36 misses, line 0 in set 1 is filled; access to address 37 then hits in line 0 of set 1; access to address 1 then hits in line 0 of set 0; access to address 38 then hits in line 0 of set 1; access to address 39 then hits in line 0 of set 1;

	Set 0						Set 1					
	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$	$C[i].tag$	$C[i].valid$	$C[i].data[0]$	$C[i].data[1]$	$C[i].data[2]$	$C[i].data[3]$
$i=0$	0	1	0	1	2	3	4	1	36	37	38	39
$i=1$	4	1	32	33	34	35		0				
$i=2$	5	1	40	41	42	43	44	0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(e) Access to address 40 misses, line 2 in set 0 is filled; access to address 41 then hits in line 2 of set 0.

Figure 8.23 An example of set-associative cache operation.

2, 4 or 8) it is feasible to choose a very effective selection policy such as LRU while limiting the cost this implies in terms of searching in parallel for a given line amongst the C_{set} possibilities. This option was moot with the fully-associative cache since there were too many possibilities to allow an efficient and cost-effective realisation.

Again consider an example set-associative cache for a byte-addressed memory where we have that $C_{lines} = 2^3 = 8$, $C_{set} = 2^1 = 2$ and $C_{words} = 2^2 = 4$ so the total cache size is 32 bytes. There are four sets each of two lines, suppose set 0 groups together lines 0 and 1, set 1 groups together lines 2 and 3 and so on. Imagine we feed it the same stream of memory accesses as before:

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41$$

and use an LRU selection policy to determine the line number within each set when data is fetched. With this architecture, we would expect an access pattern such as

$$\begin{array}{llllll}
A = 1 & A_{word} = 1, A_{set} = 0, A_{tag} = 0 & A_{line} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{set} = 0, A_{tag} = 4 & A_{line} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{set} = 0, A_{tag} = 4 & A_{line} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{set} = 1, A_{tag} = 4 & A_{line} = 0 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{set} = 1, A_{tag} = 4 & A_{line} = 0 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{set} = 0, A_{tag} = 0 & A_{line} = 0 & \rightarrow & \text{hit} \\
A = 38 & A_{word} = 2, A_{set} = 1, A_{tag} = 4 & A_{line} = 0 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{set} = 1, A_{tag} = 4 & A_{line} = 0 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{set} = 0, A_{tag} = 5 & A_{line} = 2 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{set} = 0, A_{tag} = 5 & A_{line} = 2 & \rightarrow & \text{hit} .
\end{array}$$

This matches the fortuitous case of our fully-associative cache example; we have clearly used a less costly design however. The main difference is the use of LRU to determine the line numbers within each set; the fact that each set contains 2 lines means, for example, that addresses 1 and 34 no longer conflict in this case even though before they might have mapped to the same cache line. When address 1 is accessed, none of the lines in set 0 have been used so we simply select line 0 and put this at the front of the LRU list. When address 34 is accessed, although it maps to the same set as address 0, we have (at least) one line within the set which has been used less often than line 0; so we simply select line 1 for example. As the access pattern continues in this way: consider what happens when address 35 is accessed. Translation of the address gives set 0 which contains two resident lines that were loaded when we accessed addresses 0 and 34. Parallel search of the two lines in that set matches the line loaded for 34 which is the one used for the access thus producing a cache-hit.

8.3.6 Cache Organisation

8.3.6.1 Cache Bypass

Sometimes it is attractive to perform uncached access to memory. In this case, as designers of a cache or processor, we might take one of two options. Firstly, we could provide explicit instructions (or a processor mode of some sort) that perform the access without going through the cache; this is called **cache bypass**. The second option is to provide a mechanism to turn off the cache somehow. Some processors allow this to be performed as a one-off configuration by setting a flag in the status register when powered on, other processors allow dynamic turning on and off of the cache during execution. Either way, bypassing use of the cache means that maintaining consistency between the cache and memory is difficult; it is common to leave this up to the programmer using it.

8.3.6.2 Split-Use Caches

In terms of caches, we have so far avoided the issue that they might be used to store instructions as well as data. The first issue here is that one can view the caches, like the rest of the memory hierarchy, as either a Harvard or stored program architecture. That is, one either regards access to instructions and data as separate, using separate caches for each, or as unified and hence sharing one common cache. Either approach is valid but it is common, to suit the pipelining technique described in Chapter 6, that separate caches are employed. Even from a more naive standpoint it might be attractive since separating the accesses might lessen the probability of cache pollution and contention.

Either way, this affords us some opportunity to optimise our implementation. For example, an instruction cache will not really require logic to perform writes: we do not usually write into the instructions, only updating data values once execution has begun, so the instruction cache can be significantly simpler. In addition, accesses to data and instructions exhibit different forms of locality; one might optimise the cache policies to take advantage of this knowledge so that better hit-ratios can be achieved in each case.

8.3.6.3 Split-Level Caches

It is possible to split the cache into a hierarchy just as we did with the more general memory hierarchy. In particular, it is attractive to construct the hierarchy to make a distinction between caches on the same chip as the processor, so-called **on-chip caches**, and those which are separate devices or **off-chip caches**. On-chip caches are typically faster because there is less of an interface to deal with between devices. But one can only fit so much logic on-chip, therefore off-chip caches can typically be larger. One can construct multiple levels of these on-chip and off-chip caches:

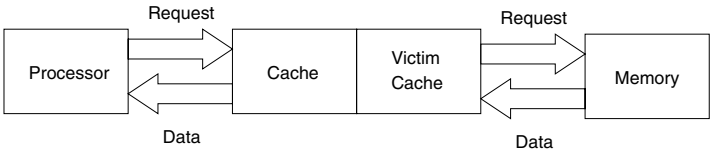


Figure 8.24 A basic block diagram of victim cache operation.

the closest level to the processor is level-one (the cache is an L1 cache), the next level in the hierarchy is level-two (the cache is an L2 cache) and so on.

This hierarchy represents a trade-off in terms of size and speed; the usual solution is to have a small on-chip cache and a larger off-chip. The hope is that the L1 cache will be able to cope with most accesses very quickly, and that it cannot deal with will mostly be dealt with by the L2 cache rather than main memory. This combined approach relies on the principle of locality to hold in order to be effective.

8.4 Advanced Cache Memories

8.4.1 Victim Caches

Considering only data caches once again, an important aspect of cache memory design is the trade-off between hit-ratio and access time. On one hand, more involved replacement policies and address translation, for example, improve hit-ratio; an improvement in hit-ratio typically means a decrease in accesses to main memory. On the other hand, as implementation of the cache design becomes more complex, the critical path of access logic becomes longer; this increases the time taken for cache hits and cache misses alike. A direct-mapped cache makes the trade-off in favour of lower access latency at the cost of lower hit-ratio; a set-associative cache improves hit-ratio using a more complex design but typically increases the access latency as a result.

Consider, for the sake of argument, a cache architecture with one level; there is a direct-mapped L1 cache between the processor and main memory. The **victim cache** design [30] makes a slightly different trade-off between hit-ratio and access time than a set-associative cache. Both hope to reduce the number of conflict misses, i.e., misses which are caused by two addresses mapping to the same cache line of the L1 cache and require access to main memory. Instead of altering the L1 cache design per se, the victim cache is placed between the two; roughly speaking, one could consider this level-one-and-a-half of the memory hierarchy. The victim cache is a small (only a few lines in size), fully-associative cache; problems with design complexity and access time are limited due to the small size. The aim is to capitalise

on the fact that empirically, it is common for cache lines that have recently been evicted from the L1 cache to be accessed again some time shortly after eviction; one can think of this as the L1 cache not quite being able to capitalise on access locality. By placing the victim cache between the L1 cache and main memory, i.e., on the fetch path of the L1 cache, it can store these victim lines temporarily and allow subsequent access to them without the need for a more expensive access to main memory. In short, the penalty associated with eviction from the L1 cache and subsequent reloading is minimised.

Figure 8.24 shows a basic block diagram of where the victim cache sits in relation to our previous discussion of cache memory. The L1 cache and the victim cache are placed close together on-chip; their access times can thus be as fast as a single processor cycle. For a given access, the L1 cache and the victim cache are searched at the same time. When a miss occurs in both caches, an access to main memory is required with the resulting cache line stored in the L1 cache only; any evicted lines are stored in the victim cache which operates according to an LRU policy. When a miss occurs in the L1 cache but the required data is located in the victim cache, the access can be completed without access to main memory; the value is taken from the victim cache by the processor and then placed in the L1 cache.

Notice that fast access time is guaranteed: since both the L1 cache *and* victim cache are on-chip and accessed in parallel, there is little difference between a hit in the L1 cache and a hit in the victim cache. Certainly they are still both at least an order of magnitude less than access to main memory. Furthermore, as long as the original empirical statement about locality holds, i.e., that recently evicted lines are often accessed again, hit-ratio moves closer to that of a set-associated cache and is certainly improved over the basic direct-mapped design. Of course this depends on the program being executed and the resulting stream of accesses to memory, but studies in the original proposal [30] show that anywhere between 10% and 70% of all misses can be removed by a sixteen line victim cache.

To give a concrete example of victim cache operation, recall where we demonstrated cache behaviours using a stream of memory accesses whose addresses were

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

We previously used an example direct-mapped cache for a byte-addressed memory where $C_{lines} = 2^3 = 8$ and $C_{words} = 2^2 = 4$ so the total cache size is 32 bytes. Using this cache, we obtained the following behaviour:

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$
$i=0$		0						0				
$i=1$		0						0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(a) Cache is initially empty.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$
$i=0$	0	1	0	1	2	3		0				
$i=1$		0						0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(b) Access to address 1 misses, line 0 is filled.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$
$i=0$	1	1	32	33	34	35	0	1	0	1	2	3
$i=1$		0						0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(c) Access to address 34 misses, line 0 evicted then re-filled via victim cache; access to address 35 then hits in line 0.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$
$i=0$	1	1	32	33	34	35	0	1	0	1	2	3
$i=1$	1	1	36	37	38	39		0				
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(d) Access to address 36 misses, line 1 is filled; access to address 37 then hits in line 1.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$
$i=0$	0	1	0	1	2	3	0	0	0	1	2	3
$i=1$	1	1	36	37	38	39	1	1	32	33	34	35
$i=2$		0						0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(e) Access to address 1 misses, line 0 evicted then re-filled via victim cache; access to addresses 38 and 39 then hit in line 1.

	L1 Cache						Victim Cache					
	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$	$C[i]_{tag}$	$C[i]_{valid}$	$C[i]_{data}[0]$	$C[i]_{data}[1]$	$C[i]_{data}[2]$	$C[i]_{data}[3]$
$i=0$	0	1	0	1	2	3	0	0	0	1	2	3
$i=1$	1	1	36	37	38	39	1	1	32	33	34	35
$i=2$	1	1	40	41	42	43		0				
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.	.

(f) Access to address 40 misses, line 2 is filled; access to address 41 then hits in line 2.

Figure 8.25 An example of victim cache operation.

$$
\begin{array}{llll}
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 38 & A_{word} = 2, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{hit} .
\end{array}
$$

Notice that addresses $0 \ldots 3$ exactly match our definition of a victim above. That is, address 1 is initially loaded into line 0 in access one before being evicted by access two and then reloaded by access six. Figure 8.25 diagrammatically details the cache behaviour when a victim cache with four lines is added; the victim cache content is shown below the L1 cache content for each step. The behaviour changes to the following:

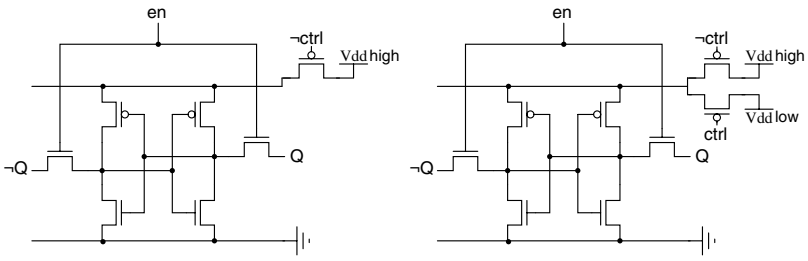
$$
\begin{array}{llll}
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{miss} \\
A = 34 & A_{word} = 2, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 35 & A_{word} = 3, A_{line} = 0, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 36 & A_{word} = 0, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 37 & A_{word} = 1, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 1 & A_{word} = 1, A_{line} = 0, A_{tag} = 0 & \rightarrow & \text{hit} \\
A = 38 & A_{word} = 2, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 39 & A_{word} = 3, A_{line} = 1, A_{tag} = 1 & \rightarrow & \text{hit} \\
A = 40 & A_{word} = 0, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{miss} \\
A = 41 & A_{word} = 1, A_{line} = 2, A_{tag} = 1 & \rightarrow & \text{hit} .
\end{array}
$$

Essentially, the penalty caused by reloading the line for addresses $0 \ldots 3$ in access six is eliminated: the reload is performed via the victim cache rather than main memory and hence we categorise the access a cache-hit rather than a cache-miss.

8.4.2 Gated and Drowsy Caches

Power consumption plays an important role in the design of embedded processors. Within a mobile telephone, lower power consumption is a good selling point; within a device such as a sensor node it is a vital operational characteristic. This has led to significant research into low-power design and manufacturing techniques. The design of low-power memory hierarchies is a specific area of interest given that memory, cache memories in particular, form a significant component in overall power consumption.

Like any transistor-based circuit, the overall power consumption in caches that use standard SRAM cells can be divided into dynamic and static components. Dy-



(a) A gated-V_{dd} cache cell. (b) A drowsy cache cell.

Figure 8.26 Two low-power memory cell designs.

namic power dissipation occurs when a logic gate switches from one state to another only; static power is dissipated by every logic gate independent of the switching activity. Originally the dynamic component dominated the overall power consumption. However, as transistors have become smaller, and hence their density has increased, static power dissipation has started to have a major impact on the overall power consumption.

There are many approaches to reduce power dissipation in cache memories; the **gated-V_{dd} cache** design [54] and the **drowsy cache** design [22, 31] are both attractive. Roughly speaking, both designs offer a mechanism to turn cache lines into a low-power state when they are not serving a useful purpose, i.e., not being regularly accessed and hence not improving memory access performance. Both designs achieve this by altering the basic SRAM cell design; the resulting cells are shown in Figure 8.26; the *ctrl* signal is used to control the cell. Memory cells in a gated-V_{dd} cache structure can be powered off using a transistor that disconnects the power supply to the cell. While this technique clearly reduces overall power dissipation, the information stored in a cell is not preserved; the design is referred to as a non-state-preserving cache. The drowsy cache offers an alternative, state-preserving technique. Memory cells in a drowsy cache structure are capable of operating in a normal mode, where they can be freely read or written to, and a drowsy, low-power mode where they retain their state but need to be woken before any access can occur. This low-power mode is facilitated by switching from being powered by the normal $V_{dd\ high}$ input to an alternative low-power voltage labelled $V_{dd\ low}$. The trade-off between the two designs should be clear: either totally power off the cells (as in the gated-V_{dd} cache design) and incur a potential performance penalty if cells need to be reloaded having been emptied of content, or retain the content (as in the drowsy cache design) and incur a power dissipation penalty because cells are only in a low-power state rather than powered off. This trade-off requires careful thought since cache misses that result from, for example, powering down gated-V_{dd} cache lines require more power than a cache hit; therefore any short-term saving from powering down such lines might be lost if they are accessed again in the longer term.

Having replaced the memory cells within a cache design, it remains to control those cells somehow so they are forced into low-power mode according to some reasonable heuristic or policy. That is, we need some control logic which detects when we could benefit from switching a given cache line into low-power mode: the more accurate this detection (in relation to the associated penalty for getting it wrong), the better our trade-off between power dissipation and performance becomes. Two fairly basic approaches can be identified; see for example [22, Section 2]:

"Simple" Maintain a global counter which is incremented after each processor cycle; when the global counter reaches some threshold, place all cache lines into low-power mode.

"No-Access" Maintain a global counter which is incremented after each processor cycle and a single bit associated with each cache line which is set when the line is accessed; when the global counter reaches some threshold, place all cache lines with their access bit not set into in low-power mode.

The "simple" policy is clearly easier to implement and probably requires less control and state logic; however, it is less accurate at powering down cache lines than the "no access" policy in the sense that it is wrong more often about lines which are not being used. More advantages and disadvantages of each policy depend on the type of cache being used, the program being executed and the resulting stream of accesses to memory, and the threshold alluded to which triggers updates.

8.5 Putting It All Together

To flesh out the components in the basic Verilog processor model developed at the end of Chapter 5, we need two main devices which are not obviously implementable using low-level logic components: a general-purpose register file and a memory system (including a cache). Both devices were initially implemented as 2-dimensional Verilog registers; we now know that there is more involved under the surface. To try to make it more obvious that the devices can be realised using techniques and components we know about, we will try to write modules that more closely model their real operational behaviour. However, we will still rely on a bit of help from Verilog: we have already showed how a 2-dimensional register can be implemented using D-type flip-flops, and continue to rely on the 2-dimensional register to store data as a result. That is, to make things simple enough to read and explain, the difference is that our devices will implement associated control logic and assume we can swap the register for the flip-flop-based memory with additional work.

```
module gpr( input  wire [ 4:0] addr_w0,
            input  wire [31:0] data_w0,
            input  wire        we0,

            input  wire [ 4:0] addr_r0,
            output wire [31:0] data_r0,
            input  wire        re0,

            input  wire [ 4:0] addr_r1,
            output wire [31:0] data_r1,
            input  wire        re1 );

  reg [31:0] data[1:31];

  assign data_r0 = re0 ? ( addr_r0 == 0 ? 32'b0 : data[ addr_r0 ] ) : 32'bZ;
  assign data_r1 = re1 ? ( addr_r1 == 0 ? 32'b0 : data[ addr_r1 ] ) : 32'bZ;

  always @( posedge we0 )
  begin
    if( addr_w0 != 0 )
      data[ addr_w0 ] = data_w0;
  end

endmodule
```

Listing 8.5 A Verilog module that models the general-purpose register file.

8.5.1 Register File

The register file is perhaps the most simple of the devices we require; it is simply a memory with two read ports and one write port. That is, we can read two values and write one value at once. In addition, we required that it implements the special behaviour associated with register number 0 which always produces the value 0 when read and discards any writes.

Listing 8.5 details a simple Verilog module for such a device. The interface includes three ports which supply an address and an enable signal to indicate that port is in operation; for example, the write port accepts the register address on `addr_w0` and is enabled by `we0`. Each interface also includes an input or output signal to convey data over depending whether the port is used to read or write data, for example the write port has `data_w0` as input. The register values themselves are held in a two dimensional Verilog register called `data`. Notice that the indexing excludes register number 0 which is not required. Two continuous assignments implement the read ports. Each assignment tests if the associated port is enabled; if it is not, the assignment writes `32'bZ` onto the output to implement 3-state style behaviour. If the port is enabled, the assignment firsts tests if register number 0 is being accessed, and outputs the value 0 as a result, or simply looks up the register value if not. Writes to the register file are edge triggered. A short process, triggered by `we0`, first checks that the write is not to register number 0 and updates the register value with the input `data_w0` if not.

```
module mem( input  wire        clk,
            input  wire        rst,

            input  wire [31:0] prev_addr,
            input  wire [ 7:0] prev_data_w,
            output reg  [ 7:0] prev_data_r,
            input  wire        prev_mrq,
            output reg         prev_mrs,
            input  wire        prev_re,
            input  wire        prev_we );

  reg [7:0] data[0:1023];

  always @ ( posedge rst )
  begin
    $readmemh( "memory/mem.bin", data, 0 );
  end

  always @ ( posedge prev_mrq )
  begin
    if     ( prev_re )
      prev_data_r = data[ prev_addr ];
    else if( prev_we )
      data[ prev_addr ] = prev_data_w;

    #6 prev_mrs = 1;
       @( negedge prev_mrq );
    #6 prev_mrs = 0;
  end

endmodule
```

Listing 8.6 A Verilog module that models main memory.

8.5.2 Main Memory

Listing 8.6 details the main memory module which closely resembles the one we developed previously: remember that we claimed that we could replace the 2-dimensional Verilog register with a concrete memory design based on D-type flip-flops. The only real difference is that we include a process that can initialise the memory content.

To initialise the memory, as triggered by a positive edge on the reset signal, we use the `$readmemh` system task. This has the effect of reading the file specified as the first argument into the 2-dimensional register specified as the second argument at an offset specified by the third argument. In this case, the file `mem.bin` is filled with 256 lines which look like

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
.. .. .. ..
```

Each value represents one byte encoded as two hexadecimal digits; using the file essentially just zeros the memory content.

```
module dmc( input  wire        clk,
            input  wire        rst,

            input  wire [31:0] prev_addr,
            input  wire [ 7:0] prev_data_w,
            output reg  [ 7:0] prev_data_r,
            input  wire        prev_mrq,
            output reg         prev_mrs,
            input  wire        prev_re,
            input  wire        prev_we,

            output reg  [31:0] next_addr,
            output reg  [ 7:0] next_data_w,
            input  wire [ 7:0] next_data_r,
            output reg         next_mrq,
            input  wire        next_mrs,
            output reg         next_re,
            output reg         next_we );

  ...

endmodule
```

Listing 8.7 The interface to a Verilog module that models a direct-mapped cache.

8.5.3 Cache Memory

To describe the cache memory, we again opt for a high-level modelling approach. Of the three designs introduced, direct-mapped caches are the most conceptually simple. To follow the example presented previously, we model such a cache parametrised to house eight lines each holding four 8-bit sub-words. The Verilog code is too long to present on one page but the interface is shown in Listing 8.7. The cache module has one interface, whose signals are prefixed with `prev`, for incoming transactions with the processor and one, whose signals are prefixed with `next`, for outgoing transactions with the memory. To operate the memory via the outgoing interface, two tasks are used; these are shown in Listing 8.8. The `ld_byte` and `st_byte` tasks implement the handshake protocol which allows the cache to load and store bytes from and into the memory. Their operation is fairly straightforward but consider the `ld_byte` task as an example. It starts by waiting for the next positive clock edge to synchronise the operation; once this is encountered it sets the address, read enable and memory request signals to initiate the transaction. The task then waits for a positive edge on the memory response signal which indicates that the memory has completed the operation; the task clears the read enable and memory request signals and reads the incoming value from the data bus. Finally it waits for the memory response signal to be cleared by the memory before continuing.

The cache operation is implemented using two processes shown in Listing 8.9. This code does not show the definitions of internal cache structures: `tag` holds the tags for the cache lines, `valid` holds the valid flags for the cache lines and `data` holds the cache lines themselves. The first process simply initialises the cache content to empty by setting the valid flags for all lines to false; this is triggered

```
task ld_byte( input  [31:0] t_addr,
              output [ 7:0] t_data );
begin
  @ ( posedge clk )
  begin
    next_addr = t_addr;

    next_re  = 1;
    next_mrq = 1;
  end

  @ ( posedge next_mrs )
  begin
    next_re  = 0;
    next_mrq = 0;

    t_data = next_data_r;
  end
  @ ( negedge next_mrs );
end
endtask

task st_byte( input  [31:0] t_addr,
              input  [ 7:0] t_data );
begin
  @ ( posedge clk )
  begin
    next_addr   = t_addr;
    next_data_w = t_data;

    next_we  = 1;
    next_mrq = 1;
  end

  @ ( posedge next_mrs )
  begin
    next_we  = 0;
    next_mrq = 0;
  end
  @ ( negedge next_mrs );
end
endtask
```

Listing 8.8 Verilog tasks to access memory via the outgoing cache interface.

by a positive edge on the reset signal. The second process is triggered every time there is a memory request via the incoming interface, i.e., a memory request from the processor. It starts by calculating the tag, line number and word offset from the incoming address. It can then check the associated line to see if it contains the required data: if the line is not valid or the stored and computed tags do not match, the line needs to be fetched from memory using the `ld_byte` task. Once the data is definitely resident in the line, we can fulfil the request, writing the data through via the outgoing interface to memory if the request was a write. Finally we instrument a small artificial delay, to model the cache latency.

It is interesting to see how the cache behaves when connected to the main memory module described above. Listing 8.10 demonstrates how the two can be instantiated to form a composite cached memory system; essentially one just has to wire

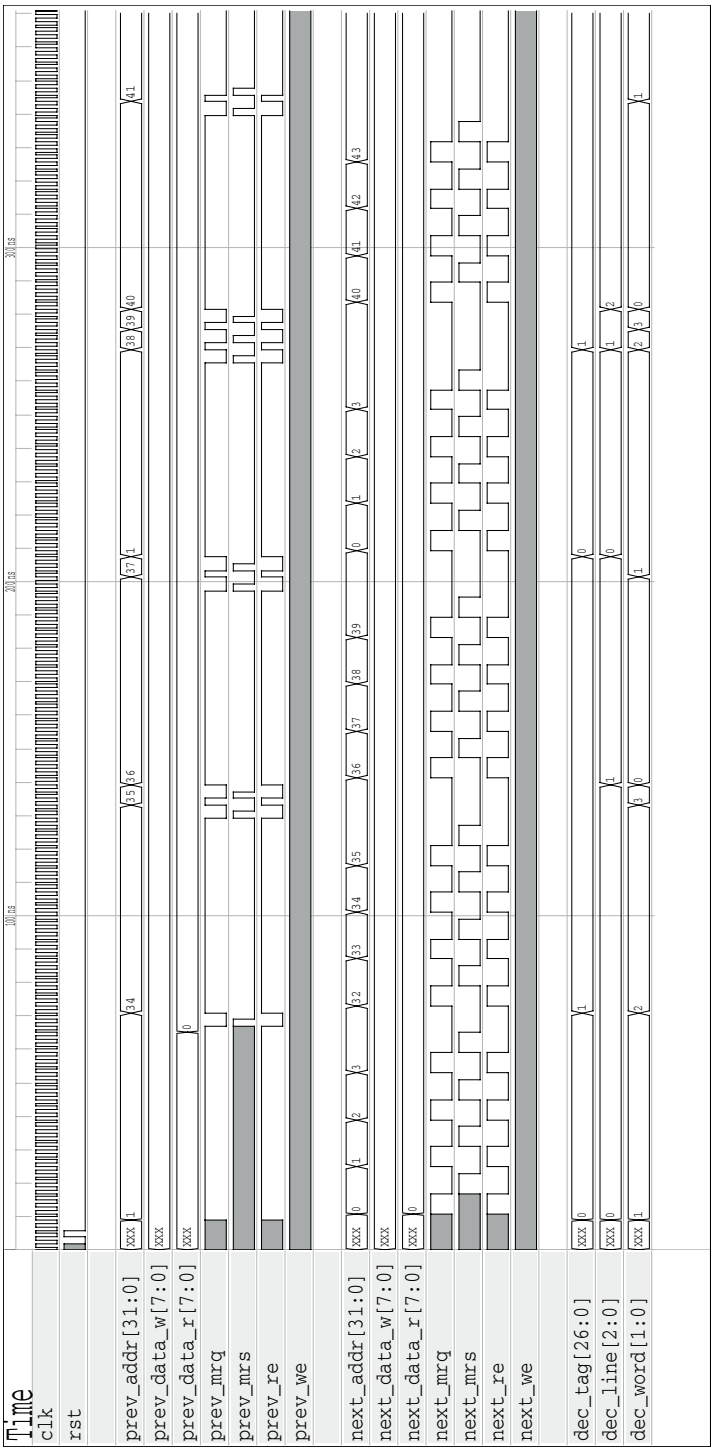


Figure 8.27 Behaviour of the composite cached memory system.

```
always @( posedge rst )
begin
  for( i = 0; i < 8; i = i + 1 )
    valid[i] = 0;
end

always @ ( posedge prev_mrq )
begin
  dec_tag  = prev_addr[31:5];
  dec_line = prev_addr[ 4:2];
  dec_word = prev_addr[ 1:0];

  if( ( !valid[dec_line] ) || ( dec_tag != tag[dec_line] ) )
  begin
    tag  [dec_line] = dec_tag;
    valid[dec_line] =       1;

    ld_byte( {dec_tag,dec_line,2'b00}, data[{dec_line,2'b00}] );
    ld_byte( {dec_tag,dec_line,2'b01}, data[{dec_line,2'b01}] );
    ld_byte( {dec_tag,dec_line,2'b10}, data[{dec_line,2'b10}] );
    ld_byte( {dec_tag,dec_line,2'b11}, data[{dec_line,2'b11}] );
  end

  if     ( prev_re )
    prev_data_r = data[{dec_line,dec_word}];
  else if( prev_we )
    data[{dec_line,dec_word}] = prev_data_w;

  if( prev_we )
    st_byte( {dec_tag,dec_line,dec_word}, data[{dec_line,dec_word}] );

  #2 prev_mrs = 1;
     @( negedge prev_mrq );
  #2 prev_mrs = 0;
end
```

Listing 8.9 Verilog processes that determine the cache behaviour.

the two devices so that they can talk to each other. To test the combined system and inspect the behaviour of the cache, we fed the same sequence of loads to it as in our more theoretical outline:

$$1, 34, 35, 36, 37, 1, 38, 39, 40, 41.$$

Figure 8.27 shows the cache behaviour during these accesses. Remember that the memory is zeroed during initialisation by the reset signal; the values read are hence all 0. The actual cache behaviour is much more interesting. We start by loading the value at address 1. The cache computes the tag as 0, the line number as 0 and the word offset as 1. Since all the cache lines are initially invalid, this causes a cache miss and the cache is forced to load the associated line, i.e., addresses 0, 1, 2 and 3, from main memory. The same pattern occurs when we try to load the value at address 34. However, when we try to access the value at address 35 the access completes much more quickly; there is no interaction with main memory at all since the value we require is already resident in line 0. The access pattern continues as

```
module cached_mem( input  wire        clk,
                   input  wire        rst,

                   input  wire [31:0] prev_addr,
                   input  wire [ 7:0] prev_data_w,
                   output wire [ 7:0] prev_data_r,
                   input  wire        prev_mrq,
                   output wire        prev_mrs,
                   input  wire        prev_re,
                   input  wire        prev_we );

  wire [31:0] next_addr;
  wire [ 7:0] next_data_w;
  wire [ 7:0] next_data_r;
  wire        next_mrq;
  wire        next_mrs;
  wire        next_re;
  wire        next_we;

  dmc t0( .clk        (clk        ),
          .rst        (rst        ),

          .prev_addr  (prev_addr  ),
          .prev_data_w(prev_data_w),
          .prev_data_r(prev_data_r),
          .prev_mrq   (prev_mrq   ),
          .prev_mrs   (prev_mrs   ),
          .prev_re    (prev_re    ),
          .prev_we    (prev_we    ),

          .next_addr  (next_addr  ),
          .next_data_w(next_data_w),
          .next_data_r(next_data_r),
          .next_mrq   (next_mrq   ),
          .next_mrs   (next_mrs   ),
          .next_re    (next_re    ),
          .next_we    (next_we    ) );

  mem t1( .clk        (clk        ),
          .rst        (rst        ),

          .prev_addr  (next_addr  ),
          .prev_data_w(next_data_w),
          .prev_data_r(next_data_r),
          .prev_mrq   (next_mrq   ),
          .prev_mrs   (next_mrs   ),
          .prev_re    (next_re    ),
          .prev_we    (next_we    ) );

endmodule
```

Listing 8.10 Instantiating the main memory and direct-mapped cache to form a composite cached memory system.

one would expect with cache-misses producing long access latencies and cache-hits completing much more quickly.

8.6 Further Reading

- J.L. Hennessy and D.A. Patterson.
  Computer Architecture: A Quantitative Approach.
  Morgan-Kaufmann, 2002. ISBN: 1-558-60724-2.
- M.D. Hill, N.P. Jouppi and G.S. Sohi.
  Readings in Computer Architecture.
  Morgan-Kaufmann, 2000. ISBN: 1-558-60539-8.
- D.A. Patterson and J.L. Hennessy.
  Computer Organization and Design: The Hardware/software Interface.
  Morgan-Kaufmann, 2004. ISBN: 1-558-60604-1.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.
- A.S. Tanenbaum.
  Structured Computer Organisation.
  Prentice-Hall, 2005. ISBN: 0-131-48521-0.

8.7 Example Questions

33. Draw a diagram of a typical memory hierarchy explaining the types of device one would expect to find in each level, their key characteristics and the principles that allow the memory hierarchy to improve system performance.

34. A given processor accesses memory using 32-bit addresses. Suppose there is a cache between the processor and a byte-addressable main memory of 512 bytes total size and with a line size of 8 bytes. For each of the following cache architectures specify how the 32-bit address would be used by the address translation mechanism in the cache:

a. Direct-mapped cache.
b. 2-way set-associative cache.
c. 4-way set-associative cache.
d. Fully-associative cache.

For example, in the direct-mapped case you would need to specify which bits of the address are used for the tag, the line number and the word offset.

35. Cache misses can be classified as either compulsory misses, capacity misses or conflict misses. For **each** class of cache miss, explain why it might occur and how changes in the cache architecture can reduce how often it occurs.

36. A given 32-bit processor has a unified level-one cache. The cache is 1 kilobyte in size, holds 4 bytes per-line and is direct-mapped. A programmer writes, compiles and executes the following program on the processor:

```
char A[1024], B[1024], C[1024];

int main( int argc, char* argv[] )
{
  for( int i = 0; i < 1024; i++ ) {
    A[i] = B[i] + C[i];
  }
}
```

a. Calculate the number of bits of an address required for each of the following quantities required to locate items in the cache:

   i. cache word address,
   ii. cache line address,
   iii. cache tag.

   and show how they are mapped onto the address.

b. Explain the following terms in relation to the cache hardware using the above program as an example:

   i. spacial locality,
   ii. temporal locality,
   iii. cache interference or contention.

c. Describe the sequence of cache hits and misses caused by accesses to `A[i]`, `B[i]` and `C[i]` as the loop progresses.

d. The definition of arrays `A`, `B` and `C` are changed to read:

```
char A[1025], B[1025], C[1025];
```

   i. Explain how this changes the sequence of cache hits and misses described above and why it improves on the original program.
   ii. Outline a different cache architecture that could increase the performance of the original program without changing.

Chapter 9
Advanced Processor Design

The key to performance is elegance, not battalions of special cases.

– J. Bently and D. McIlroy

Any sufficiently advanced technology is indistinguishable from magic.

– A. Clarke

Abstract In the context of RISC processors, almost every significant improvement in performance has come about as a result of harnessing some form of **parallelism**. Parallelism is the act of performing several small sub-tasks at the same time in order to more quickly complete a larger task; formally, this is different from the concept of **concurrency**. In this chapter, we focus almost exclusively on the concept of **Instruction-Level Parallelism (ILP)** which allows instructions from within a single program to execute in parallel. The advantage of this approach is demonstrated by three main design techniques: **pipelined processors**, **superscalar processors** and **vector processors**.

9.1 Introduction

The types of parallelism that exist within processor designs are usually categorised as being related to either data or instructions. That is, we typically want to operate either on several items of data at the same time using a single instruction, or execute several different instructions at the same time. Our focus is mainly expanding upon the non-parallel concepts presented previously; as a result it is worthwhile looking briefly at the broader subject of parallel computer architecture so as to fix some ideas and notation. The basic idea that runs throughout this chapter is that a single processor should be composed of multiple **Processing Elements (PEs)**, each of which operates in parallel and can potentially communicate with each other via some mechanism. Our initial aim is to present a high-level outline of how these

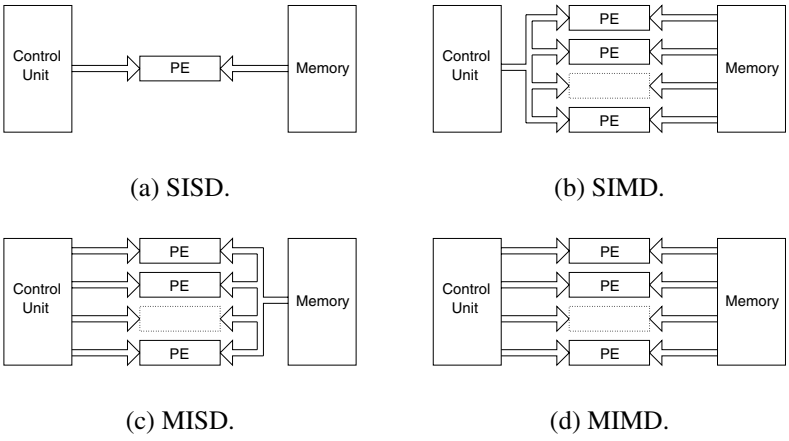


Figure 9.1 A graphical representation of four parallel computer categorisations.

components could be organised and used to execute instructions in parallel, and then to introduce concrete processor designs which capitalise on said organisations.

9.1.1 A Taxonomy of Parallelism

9.1.1.1 Computational Model

The **computational model** of a parallel computer categorises the different instruction and data streams within the processor. That is, given the processor will be operating on some data using some instructions, it outlines the behaviour of a given PE in relation to the other PEs. The canonical summary is **Flynn's Taxonomy**, a categorisation introduced by Michael Flynn in 1972 [24] which is described diagrammatically in Figure 9.1:

Single Instruction, Single Data (SISD) There is only one stream of instructions and one stream of data; ignoring co-processors and additional PEs added to produce identical, redundant computation (e.g., to cope with errors), there can only be one useful PE in such a processor. Examples of SISD computers include that described in Chapter 5.

Single Instruction, Multiple Data (SIMD) There is only one stream of instructions but there are many streams of data; thus we can have many PEs each executing the same instruction at a given point in time but using different data in each case. Examples of SIMD computers include the Cray series of vector supercomputers and the Graphics Processing Units (GPUs) that render 3D images within most modern graphics cards.

Multiple Instruction, Single Data (MISD) It is hard to justify the existence of MISD processors, in the sense that it is hard to see how such a device could compute useful results: what we are describing is a situation where we take a single data stream and operate on it using many different instruction streams. Examples of MISD processors are rare but application areas are typically those that require some form of duplicate computation. For example, a computer to forecast weather might run several simulation models on the same dataset and average the results to get a more reasonable result.

Multiple Instruction, Multiple Data (MIMD) There are many streams of instructions and many streams of data; essentially we have many PEs executing different instructions on different data at the same time. In a sense, this is the most general form of parallel processor. Examples include a cluster of workstations (where the PEs are physically separate computers) connected by a network in order to collectively perform some task; the Beowulf design developed by Donald Becker at NASA is an example of how one can construct such system inexpensively. However, recent advances in processor design have led us to so-called multi-core designs whereby one has several processor cores on one microchip; the combined system is essentially a small-scale MIMD computer.

This taxonomy is usually extended to include the so-called **Single Program, Multiple Data (SPMD)** model. This is similar to SIMD in the sense that each processor is presented with a single set of instructions to execute. However, unlike SIMD a given PE within an SPMD processor can obtain an identifier (i.e., a processor number of some sort) and make conditional decisions based on that identifier. So, for example, PE number one might take one control-flow path within the single program while PE number two might take another. Hence although they are both given a single program, they might execute different instructions within it.

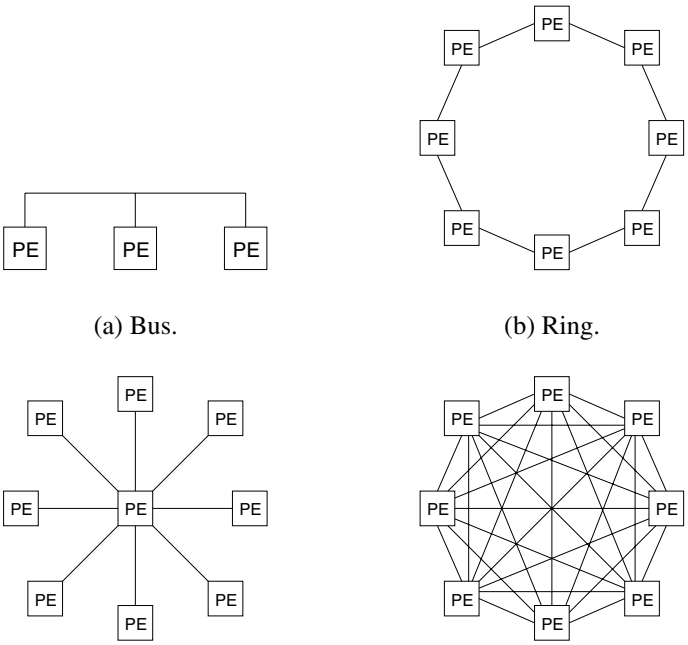
9.1.1.2 Communication Model

The **communication model** of a parallel processor categorises how the independent PEs communicate with each other. Although there are variations on each, roughly speaking this comes down to only two main choices:

Distributed Memory Each PE has a dedicated local memory; PEs are linked together using some sort of network interconnect. Since intermediate results are communicated via the network in some way, this gives rise to the term **message passing architecture**. That is, intermediate results are passed between the PEs as messages over the network rather than written into memory by one PE and then read by another.

Several systems exist to allow easy development of distributed memory parallel programs; MPI [49] is an industry standard example.

Shared Memory All PEs can load from and store to the same set of addresses, i.e., there is a shared address space. In concept this means there is a single physical memory that has many read and write ports, one interface for each PE. How-



(a) Bus. (b) Ring.

(c) Star. (d) Fully connected.

Figure 9.2 A graphical representation of four interconnection networks.

ever, this is often difficult to realise so the address space can be mapped over a number of physical memories; that is, each PE maintains a local memory which is tasked with holding a sub-set of the total memory content. In this case some mechanism must be provided to access potentially non-local memory locations.

A processor that offers a shared address space communication model is often called a **multiprocessor** and includes a new generation of multi-core processors. In this context, processors where access to each location in the address space takes the same time are said to have **Uniform Memory Access (UMA)**; where this does not apply they have **Non-Uniform Memory Access (NUMA)**. Where the single address space is mapped over a number of physical memories, one can see that NUMA is quite probable: local accesses will be faster than non-local accesses.

Cache memories can cause a problem in shared memory processors if the address space is mapped over several memories. Specifically, if one memory holds stale data, i.e., the real value is held in a local cache for that memory, a remote PE accessing that data might get the wrong value unless it also accesses the memory through the cache. A hardware-based **cache coherency** protocol attempts to solve this problem; roughly speaking a PE sneaks a look at the local cache to

decide where it should read the correct value from. Alternatively this problem can be delegated to the program to manage themselves.
Several systems exist to allow easy development of shared memory parallel programs; OpenMP [15] is an industry standard example.

The means by which these different mechanisms are implemented is crucial and quite subtle in terms of performance. For example, it is important to realise that the network interconnect that links the PEs in a distributed memory system influences how easy it is to pass messages around. Figure 9.2 shows just four of many different configurations and demonstrates the trade-offs involved. For example, the bus-based network is simple to construct and it is easy to add new PEs; however, only one pair can communicate at a time which potentially imposes a performance bottleneck. In contrast, the fully connected network is complex to construct and hard to add new PEs; however, any pair can communicate with each other at any time.

9.1.2 Instruction-Level Parallelism (ILP)

The previous section described quite a general spectrum of parallel computer architectures. In particular, one might view designs within Flynn's Taxonomy as a means to execute arbitrary programs in parallel with each other. In reality, it is useful to work at the level of instructions rather than programs. As such, it is common to design and implement mechanisms that allow individual instructions to be executed in parallel. Put rather too simply, one can imagine some mechanism inside a processor which examines instructions in the program and for those which can be executed in parallel, achieves some performance improvement by doing so.

The order that instructions go into the processor is termed the **program order**, which is the order that they would be executed on a sequential processor. However, if one considers a processor that exploits parallelism, the execution order might differ from the program order; we use the term **data order** in this case. The analysis of instructions to determine which can be executed in parallel is termed the **scheduling** of instructions. Scheduling techniques exist somewhere between two extremes: **static scheduling** mainly exploits ILP at compile-time whereas **dynamic scheduling** mainly exploits ILP at run-time. More simply, one can place responsibility on the programmer and force the program to make parallelism explicit (i.e., *it* tells the processor which instructions to execute in parallel). Alternatively, one can place responsibility on the processor and have it automatically detect and exploit implicit parallelism within the program.

However, there are several generic barriers which all ILP-based processors must consider and deal with to ensure a given program executes correctly; we describe these in detail below.

9.1.2.1 Consistency

As discussed previously, a processor ISA can be viewed as a contract that guarantees execution of a given program is consistent with a set of defined semantics. More simply, if one executes a program on some processor then it should execute as we expect it to. To capture the idea of a program executing consistently with the semantics, one can define two features:

Definition 37. The term **program consistency** refers to the order in which execution of instructions is completed. For example, consider executing an addition instruction X in parallel with a division instruction Y. If Y generates an exception as the result of dividing by zero, the processor must halt execution and perform some remedial action (e.g., set a flag in the status register). However, depending on which of X and Y was specified first in program order, the current processor state might not be as expected. In particular, if Y was specified after X and then generates the exception, the current state might look like Y was specified before X which would be inconsistent with the program order.

Definition 38. The term **memory consistency** refers to the order in which accesses to memory are performed. For example, consider executing two store instructions X and Y in parallel; either X or Y could be the last instruction to access memory and hence store a value. Since either X or Y was specified first in program order, the end result in memory can be inconsistent with that order.

Both the above can be further described as either **strongly consistent** or **weakly consistent**. Strong processor consistency means instructions must complete execution **in-order**, i.e., in program order; weak processor consistency means that instructions can complete execution **out-of-order**, i.e., different to program order. Strong memory consistency means that the order of memory accesses must match the instructions that invoke them; weak memory consistency means that the memory accesses can be re-ordered as long as they respect dependencies between them.

9.1.2.2 Dependencies

Even considering a case where program and memory consistency is guaranteed somehow, it should be clear that not all instructions in a program can be executed in parallel with each other. Imagine instruction X comes before instruction Y in the program order; we can categorise the situations where one instruction is **dependent** on another as follows:

Resource Dependency A resource dependency is more to do with the micro-architecture than the program, but captures the idea that some resources cannot be shared. In particular, if two instructions X and Y require access to the same resource, then they cannot be executed at the same time. For example, consider the program fragment:

$$
\begin{aligned}
&\ldots\\
&GPR[2] \leftarrow GPR[3] \cdot GPR[4]\\
&GPR[5] \leftarrow GPR[6] \cdot GPR[7]\\
&\ldots
\end{aligned}
$$

Both instructions #0 and #1 require the use of a multiplier to compute their results; if there is only one multiplier available, the two instructions cannot be executed in parallel.

Data Dependency If the result of instruction X is required as an input to instruction Y, Y depends on X. This is the intuitive definition of dependency: if instruction X calculates some value that Y needs, then clearly Y cannot be executed until X has calculated said value. The relationship is transitive in the sense that if instruction Y depends on X and some third instruction Z depends on Y, Z depends on X as well. For example, consider the program fragment:

$$
\begin{aligned}
&\ldots\\
&GPR[2] \leftarrow GPR[3] + GPR[4]\\
&GPR[5] \leftarrow GPR[2] + GPR[2]\\
&GPR[6] \leftarrow GPR[5] + GPR[5]\\
&\ldots
\end{aligned}
$$

Instruction #1 depends on the result produced by instruction #0, also instruction #2 depends on the result produced by instruction #1. In this case, for example, instruction #1 cannot be executed until the result produced by executing instruction #0 is available. Where the dependency is based on a value loaded from memory, we often use the term **load-use dependency**, whereas if the value is generated by some other means (for example an arithmetic instruction) we use the term **define-use dependency**.

Name Dependency There are two types of name dependency; both types are often described as "fake" data dependencies in the sense that they both look like a normal data dependency, but can be removed by a process of renaming. A so-called **anti-dependence** exists if instruction Y writes to a register that X reads from, ordering of X and Y must be preserved to ensure X reads the correct value. For example, in the program fragment

$$
\begin{aligned}
&\ldots\\
&GPR[2] \leftarrow GPR[3] + GPR[4]\\
&GPR[3] \leftarrow GPR[4] + GPR[5]\\
&\ldots
\end{aligned}
$$

instruction #0 reads from $GPR[3]$ but instruction #1 writes to $GPR[3]$; as written, instruction #1 depends on instruction #0 in the sense that for the program to have the right meaning, instruction #1 cannot write a new result to $GPR[3]$ until instruction #0 has read the previous one. This dependency can be broken by rewriting the program so that instruction #1 writes into a different register. For, example in the program fragment

$$
\begin{array}{l}
\ldots \\
GPR[2] \leftarrow GPR[3] + GPR[4] \\
GPR[6] \leftarrow GPR[4] + GPR[5] \\
\ldots
\end{array}
$$

instruction #1 is no longer dependent on instruction #0. The related concept of **output dependency** exists if instruction X writes to the same register as instruction Y, ordering of X and Y must ensure the end result is correct. For example, in the program fragment

$$
\begin{array}{l}
\ldots \\
GPR[2] \leftarrow GPR[3] + GPR[4] \\
GPR[5] \leftarrow GPR[2] + GPR[2] \\
GPR[2] \leftarrow GPR[3] - GPR[4] \\
\ldots
\end{array}
$$

instructions #0 and #2 both write to $GPR[2]$ so in some sense they are dependent on each other; instruction #0 has to be executed first in order for the program to have the right meaning. Again this dependency can be broken by rewriting the program so that instruction #2 writes into a different register, for example

$$
\begin{array}{l}
\ldots \\
GPR[2] \leftarrow GPR[3] + GPR[4] \\
GPR[5] \leftarrow GPR[2] + GPR[2] \\
GPR[6] \leftarrow GPR[3] - GPR[4] \\
\ldots
\end{array}
$$

Control Dependency Imagine there are a number of branch instructions in the program and X is one of them; there is a control dependency between instructions X and Y if control-flow reaching Y depends on the execution of X. In the program fragment

$$
\begin{array}{rl}
 & \ldots \\
 & GPR[2] \leftarrow GPR[3] + GPR[4] \\
 & \text{if } GPR[2] \neq 0, PC \leftarrow skip \\
 & GPR[2] \leftarrow 1 \\
skip: & \ldots
\end{array}
$$

instruction #1 is a branch which conditionally alters control-flow. Since the branch target depends on the value in $GPR[2]$, instruction #2 is control dependent on instruction #1: the outcome of executing instruction #1 decides if control-flow reaches instruction #2.

9.1.2.3 Hazards

A **hazard** is a condition that needs to be resolved in some way to ensure that an ILP-based processor avoids incorrect execution of a program as a result of instruc-

tion dependencies. Note that this issue is intrinsically tied to the micro-architecture: on one data-path a given dependency between instructions might cause a hazard condition to occur whereas on another data-path the same dependency might not cause a hazard. As a trivial example, consider our original, sequential data-path executing the program fragment

$$\begin{aligned}&\ldots\\&GPR[2] \leftarrow GPR[3] + GPR[4]\\&GPR[5] \leftarrow GPR[2] + GPR[2]\\&GPR[6] \leftarrow GPR[5] + GPR[5]\\&\ldots\end{aligned}$$

In the case where each instruction is executed sequentially, there is no hazard: the data dependency between instructions #1 and #0 does not matter because instruction #0 will always be executed before #1. In the case of a data-path which executes instructions in parallel however, this might result in a hazard if the data-path cannot guarantee the same ordering.

For data and name dependencies at least, there is a somewhat standard method for categorising hazards that can occur. Again, throughout we imagine instruction X appears before instruction Y in the program order:

Read-After-Write (RAW) A RAW hazard occurs when instruction Y reads a value from register (or memory location) R before instruction X has written it. Unless the hazard is avoided, Y gets the old, wrong value in R; one might say Y is RAW-dependent on X.

Write-After-Read (WAR) A WAR hazard occurs when instruction Y writes a value to register (or memory location) R before instruction X has read it. Unless the hazard is avoided, X gets the new, wrong value in R; one might say Y is WAR-dependent on X.

Write-After-Write (WAW) A WAR hazard occurs when instruction Y writes a value to register R (or memory location) before instruction X writes a value to the same place. Unless the hazard is avoided, the writes happen in the wrong order and the value after they finish is wrong; one might say Y is WAW-dependent on X.

9.2 Pipelined Processors

Production or assembly lines in the context of manufacturing offer a neat analogy for the concept of **pipelining** in digital logic circuits. The idea of a production line is for the end result to be produced as the combination of a number of basic steps. However, instead of performing the steps sequentially each worker within the production line can be operating on a different item at the same time. This offers two main benefits. Firstly, it allows a degree of specialisation; no longer does one person need to be expert in all stages of manufacture or require the equipment to carry out

all stages. Each worker in the production line can be dedicated to one stage, mastering it to an extent which meant their output is higher than normal. Secondly, it allows for division of labour in the sense that one can scale the level of output closely with the number of workers; roughly speaking, the more workers one has the greater the output. This is largely because of the fact that the workers in a production line can all be working effectively at the same time. In particular, one worker is not idle while waiting for another to finish or for tools to become available.

Although probably not the first to employ such a process, the manufacture of cars within the Ford Motor Company is a good example. Ford, under direction of the owner Henry Ford, used a system of continuous production lines to build automobiles. While one person was assembling the engine to car number one, another could be attending to the body work on car number two while yet another could be fitting the wheels to car number three. By around 1913, Ford had his production line down to such a fine art that they were able to double the output of all their competitors, selling half of all automobiles purchased in the USA. Although assigning each worker a dedicated task reduced accident and wasted time through their wandering around the factory, the fact that they stood in the same place for long periods performing repetitive tasks meant that RSI-type injuries were common. Ford combated the resulting high turnover of staff by increasing wages to $5 a day, cutting shift lengths to eight hours a day and installing a dedicated medical department. Productivity soared and the cost of producing each vehicle decreased as a result.

Fortunately, pipelining digital logic circuits does not suffer from these social related problems: our logic gates never tire, get RSI or complain about wages. Other than this, the principles are almost exactly the same. A basic pipelined circuit consists of a number of **pipeline stages** arranged in sequence; each will perform part of the computation. We feed inputs into the first stage and outputs are produced by the last stage. The progression of intermediate results or work through the pipeline is typically governed by one of two techniques:

Synchronous Pipeline In a synchronous pipeline, a global clock is in control. On each clock edge we say the pipeline **advances**; the stages pass their completed work onto the next stage and receive new work from the previous stage. We term the interval between one advancement and another a **pipeline cycle** in a similar way that a clock cycle is the interval between one positive or negative edge and another.

In this setting, unless each stage takes the same amount of time there is some slack where one or more stages will be idle. For example, if the clock period of a synchronous pipeline is 10ns and one of the stages takes 5ns to complete its work, it will be idle for 5ns after working before another clock pulse causes the pipeline to advance. As such, we ideally want to balance the stages so they all take roughly the same length of time to complete their work and thus minimise idleness.

Asynchronous Pipeline In an asynchronous pipeline, the aim is to remove the need for global control over when the pipeline advances and hence remove the need to manage a global clock. Roughly speaking, control is devolved into the

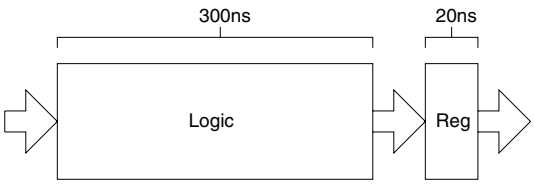


Figure 9.3 A sequential block of combinatorial logic.

pipeline stages themselves: for one stage to advance, it must engage in a simple handshake with the preceding and subsequent stages to agree when to advance. This is advantageous in that stages can operate as fast or slow as their workload, rather than the clock, dictates. For example, in the execution phase an addition may take less time to complete than a multiplication. The asynchronous pipeline can advance whenever the result is ready rather than being pessimistic and forcing advancement at the rate of the slowest operation (i.e., latency of the multiplication). However, although the global clock is removed one potential disadvantage of this approach is overhead in provision of the handshake mechanism that has to exist between stages.

Although there are positive and negative arguments for both approaches, we consider only synchronous pipelines from here on. We can measure the performance of such a pipeline using two standard metrics:

Definition 39. The **latency** of a pipeline is the total time elapsed before a given input is operated on to produce an output. This is simply the sum of the times taken by each pipeline stage.

Definition 40. The **bandwidth** or **throughput** of a pipeline is the rate at which new inputs can be offered to the pipeline (resp. outputs produced by the pipeline). This is typically limited by the stage that takes the longest time since this stage will dictate how fast the pipeline can be advanced.

When one stage passes on a result to the next, it needs to save it in a **pipeline register** (conceptually like a small storage area) rather than just driving it onto a wire. If it did not do this, half way through performing the work it might cause the wire value to change; the next stage using this value would therefore potentially use the wrong value. By using a register, the value used by the next stage is guaranteed to remain fixed until it is updated when the pipeline advances. Each pipeline register takes some time to operate, so adds to the total pipeline latency. Therefore, pipeline latency and throughput represent something of a trade-off: too many stages means lots of pipeline registers and a higher latency, too few stages means not enough parallelism is taken advantage of, lower throughput and potentially a more unbalanced pipeline.

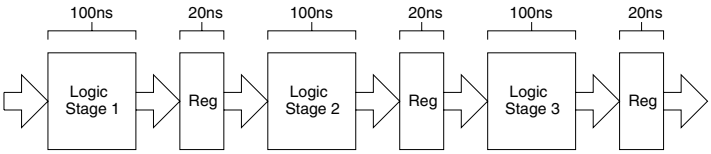


Figure 9.4 A pipelined version of the combinatorial logic block.

Consider the example in Figure 9.3 where we have a block of combinatorial logic; you can think of this as a shifter circuit perhaps. We want to improve the performance of this circuit by pipelining it so split it into three stages as shown in Figure 9.4. In both cases the latency associated with each stage or pipeline register is shown above it. The total latency of the sequential circuit is

$$300\text{ns} + 20\text{ns} = 320\text{ns}.$$

The circuit produces an output every 320ns so the rate at which the circuit produces outputs is $1\text{s}/320\text{ns}$ or around $3.12 \cdot 10^6$ outputs per-second. Looking at the pipelined circuit, we find that the total latency has increased due to the addition of several pipeline registers:

$$100\text{ns} + 20\text{ns} + 100\text{ns} + 20\text{ns} + 100\text{ns} + 20\text{ns} = 360\text{ns}.$$

At face value this seems bad: each input will actually take longer to be operated on to produce an output than in the sequential case ! However, with the pipelined circuit we can offer a new input every 120ns since this is the length of the longest stage (including the pipeline register). So the new throughput is $1\text{s}/120\text{ns}$ or $8.33 \cdot 10^6$ outputs per-second, between a two and three-fold improvement.

This example highlights a further issue in terms of pipeline design. Since we will inevitably increase the total latency by the act of pipelining, our goal is a high throughput. However, this is only useful in practise if we can keep the pipeline full. That is, if our pipeline can potentially accept a new input every 10ns but we only offer one every 50ns, it will be idle for periods of time and so the useful throughput is much lower than the actual throughput. Thus it only really makes sense to trade throughput for latency if we can guarantee the pipeline utilisation will be high, i.e., the pipeline will be full most of the time. Otherwise you might as well use a sequential circuit given one-off output will be generated quicker.

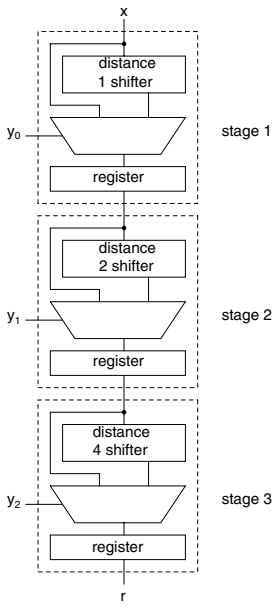


Figure 9.5 Design of a pipelined 8-bit logarithmic shifter architecture.

9.2.1 Pipelined Circuits

In the previous section we presented an abstract description of how the act of pipelining works: we now turn our attention to a simple but concrete example. We will pipeline the 8-bit logarithmic shifter from Chapter 7 that was implemented in Listing 7.6. It is easy to see how this can be split into three small stages since they are almost implicit in the design from Figure 9.5; more complex circuits are harder to split up.

Listing 9.1 details the implementation of the first pipeline stage; the other stages are very similar and are left out for the sake of brevity. The stage looks almost identical to the first part of our sequential shifter implementation; to recap, we have a multiplexer `t0` that generates the correct result given the shift type and another multiplexer `t1` that decides if we take that shifted result or the original non-shifted input. The first difference is that each stage only needs 1-bit of the shift distance `y` rather than the whole value. The major difference is that we have to include some logic to advance the pipeline stage. As such, on each positive edge of the clock signal `clk` the stage takes the computed result `w1` which comes out of the second multiplexer and assigns it into the stage result `r`.

We combine the three pipeline stages together in Listing 9.2. Again this is a simple step, we just instantiate each shifter stage and join them together using the

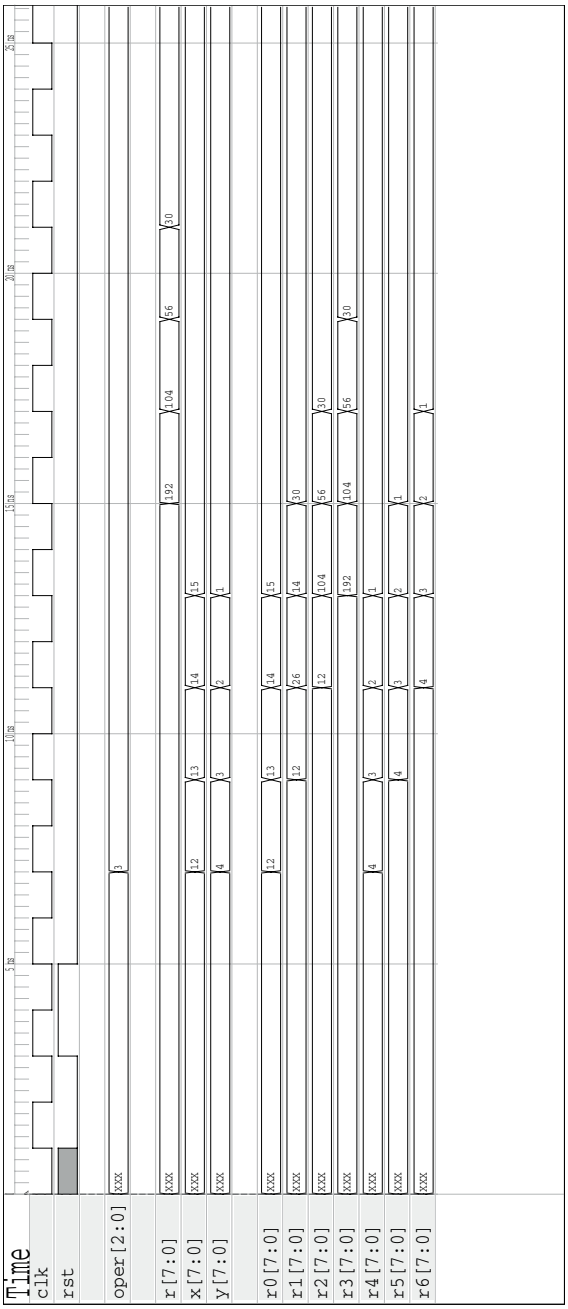


Figure 9.6 Behaviour of a pipelined 8-bit logarithmic shifter.

```
module shf_8bit_stage1( input  wire       clk,
                        input  wire       rst,

                        input  wire [2:0] oper,

                        output reg  [7:0] r,
                        input  wire [7:0] x,
                        input  wire       y );

  wire [7:0] w0;
  wire [7:0] w1;

  mux8_8bit t0( .r(w0),

                .i0({ 1'b0,        x[7:1] }),
                .i1({ {1{r[7:7]}}, x[7:1] }),
                .i2({ x[0:0],      x[7:1] }),
                .i3({ x[6:0],      1'b0   }),
                .i5({ x[6:0],      x[7:7] }),

                .s0(oper[0]), .s1(oper[1]), .s2(oper[2]) );

  mux2_8bit t1( .r(w1), .i0(x), .i1(w0), .s0(y) );

  always @ ( posedge clk )
  begin
    r = w1;
  end

endmodule
```

Listing 9.1 Implementation of the first shifter pipeline stage.

pipeline registers `r4` to `r6`. Note though that we need to include registers for the intermediate shifted results and the shift distance since, for example, stage one might be shifting a value x_1 by a distance y_1 while stage two is operating on x_2 and y_2: if we do not use pipeline registers for both quantities we could be shifting both x_1 and x_2 by y_2 which would be incorrect. At the bottom of the module we also include some logic to provide inputs to the pipeline and grab outputs. Each time the clock ticks to advance the pipeline, we assign the result from the last stage `r3` to `r`, the output for the whole shifter. We also take the two inputs `x` and `y` and assign these to the two input registers `r0` and `r4` which feed the first stage.

Figure 9.6 outlines how this design behaves; the timeline details the execution of three logical left-shift operations for the values $12_{(10)}$,$13_{(10)}$,$14_{(10)}$ and $15_{(10)}$ by the distances $4_{(10)}$, $3_{(10)}$, $2_{(10)}$ and $1_{(10)}$ bits respectively. After a few clock cycles, the `oper` signal is set to the left-shift value and the first pair of values, $12_{(10)}$ and $4_{(10)}$, are driven onto the inputs of the shifter. Note though that the result does not immediately appear since after the first clock cycle they will only have been processed by the first stage of the pipeline as shown by the `r1` signal which holds this output. Instead, we are free to send another pair of values, this time $13_{(10)}$ and $3_{(10)}$, as input. Now we have a case where stages one and two are operating in parallel on two different operations: stage one is dealing with shifting $13_{(10)}$ left by $3_{(10)}$ bits while stage two is dealing with shifting $12_{(10)}$ left by $4_{(10)}$ bits. This continues

```
module shf_8bit( input  wire       clk,
                 input  wire       rst,

                 input  wire [2:0] oper,

                 output reg  [7:0] r,
                 input  wire [7:0] x,
                 input  wire [7:0] y );

  reg  [7:0] r0;
  wire [7:0] r1;
  wire [7:0] r2;
  wire [7:0] r3;
  reg  [7:0] r4;
  reg  [7:0] r5;
  reg  [7:0] r6;

  shf_8bit_stage1 t0( .clk ( clk   ),
                      .rst ( rst   ),

                      .oper( oper  ),

                      .r   ( r1    ),
                      .x   ( r0    ),
                      .y   ( r4[0] ) );

  shf_8bit_stage2 t1( .clk ( clk   ),
                      .rst ( rst   ),

                      .oper( oper  ),

                      .r   ( r2    ),
                      .x   ( r1    ),
                      .y   ( r5[1] ) );

  shf_8bit_stage3 t2( .clk ( clk   ),
                      .rst ( rst   ),

                      .oper( oper  ),

                      .r   ( r3    ),
                      .x   ( r2    ),
                      .y   ( r6[2] ) );

  always @ ( posedge clk )
  begin
    r6 = r5;
    r5 = r4;
    r4 =  y;
    r0 =  x;
    r  = r3;
  end

endmodule
```

Listing 9.2 Implementing the whole pipelined shifter.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	$\cdots$
Fetch	#0				#1				#2	$\cdots$
Decode		#0				#1				$\cdots$
Execute			#0				#1			$\cdots$
Write				#0				#1		$\cdots$

Table 9.1 An execution timeline for a processor using a sequential data-path.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	$\cdots$
Fetch	#0	#1	#2	#3	#4	#5	#6	#7	#8	$\cdots$
Decode		#0	#1	#2	#3	#4	#5	#6	#7	$\cdots$
Execute			#0	#1	#2	#3	#4	#5	#6	$\cdots$
Write				#0	#1	#2	#3	#4	#5	$\cdots$

Table 9.2 An execution timeline for a processor using a pipelined data-path.

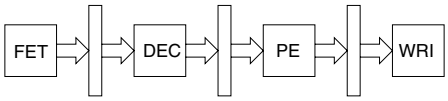


Figure 9.7 A conceptual diagram of a pipelined processor.

until a while later when the first output appears on `r` having passed through all three stages; after this we get a new result *every* clock cycle as the subsequent values we sent as inputs are completed. Notice the trade-off between throughput and latency here. Our results take longer than one clock cycle to pass through the entire circuit but once the pipeline is full, we get one result every clock cycle. The critical path of the circuit is less than the fully parallel version since the longest path is now through only one pipeline stage instead of all three; we can probably clock the circuit faster as a result. The trade-off is that it probably takes more logic to implement our pipelined design since we need to, for example, manage the pipeline registers.

9.2.2 Pipelined Processors

9.2.2.1 Basic Concept

The processor data-path we developed previously in Chapter 5 is, in the most basic sense, a logic circuit. Although it should be obvious we can pipeline components like the ALU, we can use exactly the same techniques to pipeline the processor

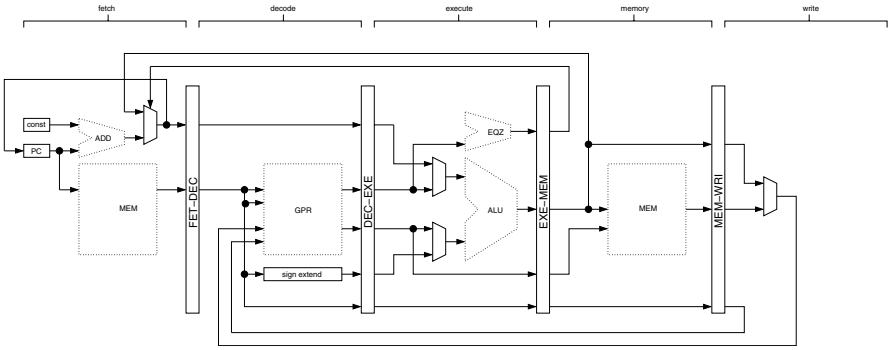


Figure 9.8 A 5-stage pipelined data-path.

data-path itself. The key thing to grasp when considering pipelined processors is that we can have more than one instruction **in-flight** at a time: this is the concept of ILP being put to practical use. To see why this might be the case, consider Table 9.1 which details an execution timeline for our original sequential data-path. In cycle t_0 we fetch instruction #0 which is then decoded in cycle t_1, executed in cycle t_2 and finally writes results in cycle t_3; we then fetch instruction #1 in cycle t_4 and so on. Now imagine we split the sequential data-path into four pipeline stages, one each for fetch, decode, execute and write. A conceptual diagram of how this would look, which is used as a standard starting point for further discussion, is shown in Figure 9.7. With such a design the fetch stage, for example, can be fetching instruction #2 while the decode stage is dealing with instruction #1 and the execute stage is dealing with instruction #0 and so on. This is highlighted in Table 9.2. With all stages working in parallel with each other, we execute around four times as many instructions in the same period; clearly this depends on factors previously introduced such as the requirement to keep the pipeline full and further complications we will introduce later.

To make this scheme possible we need to design a pipelined data-path which is capable of splitting the original into stages, each of which can operate independently on a different instruction. Figure 9.8 describes the "classic" way to do this; the design is composed of five stages in total: the fetch, decode, execute and write already described, together with a dedicated memory access stage. There are several ways this basic design can be improved but it offers a good, simple starting point. For brevity, we denote these stages FET, DEC, EXE, MEM and WRI. In the diagram, notice that the stages are separated by pipeline registers just like in our basic pipelined circuit; we denote these registers $FET-DEC$, $DEC-EXE$, $EXE-MEM$ and $MEM-WRI$. The registers hold several different values even though we have drawn them as a single block. When we need to, we subscript the pipeline register with the conventional register name from the sequential data-path. So for example $FET-DEC_{PC}$ is the value of the program counter register held

in pipeline $FET - DEC$, the value of the instruction register at $DEC - EXE$ is $DEC - EXE_{IR}$.

9.2.2.2 Control Strategy

By and large the pipeline stages perform similar roles as the steps performed by the sequential data-path, they just operate in parallel. To expand on this, we describe the operation of each stage:

Fetch (FET) The fetch stage uses its dedicated memory interface to load the next instruction to be executed; it uses the address stored in the *PC* register which is also located within the fetch stage. The *PC* is either incremented using an adder or, in the case of a taken branch, updated using values fed in from elsewhere. When the pipeline advances, the instruction to be executed and also the current value of *PC* are passed to the next stage by storing them in $FET - DEC$.

Decode (DEC) The decode stage takes the instruction from $FET - DEC$, decodes it to recover register addresses and immediate values and uses them to generate operands for the execute stage. It does this by loading operands from *GPR* and by sign extending any immediates. When the pipeline advances, these operands together with the instruction and current value of *PC* are passed to the next stage by storing them in $DEC - EXE$. The act of moving instructions into phases of actual execution is sometimes called the **issue** of instructions; a design using this strategy is termed **single-issue** in the sense that only one instruction passes into the execution stage at a time.

Execute (EXE) Having taken opcode and operands from $DEC - EXE$, the execute stage fills a similar role as in the sequential data-path. It simply invokes the ALU to work on the operands and produce some result. Unlike the diagram, the ALU may be notionally split into two parts since these operate at the same time: the main ALU performs arithmetic and logic operations while a secondary unit performs comparisons. The ALU is fed inputs from either the main operands in $DEC - EXE$ or secondary operands such as the current *PC* value which is also stored in $DEC - EXE$; multiplexers are used to make this selection which allows for the increment of *PC* for example. When the pipeline advances, generated results are stored in $EXE - MEM$.

Memory (MEM) The memory access stage is simple. It takes the instruction that is stored in $EXE - MEM$ and, if it is a memory access instruction (i.e., a load or a store), performs the corresponding operation. Notice this requires that the memory access stage has a dedicated interface to the memory, and that the stage now generates two intermediate results which, along with the instruction, are stored in $MEM - WRI$ when the pipeline advances. In this particular design, the memory access stage also feeds the branch condition and computed *PC* value back to the fetch stage should a branch be taken.

Write (WRI) Finally, the write stage takes any intermediate results which have been generated, either by execution of some operation or the loading something from memory, and writes them into the general-purpose register file. Unlike most

other signals in the pipeline that are forward facing, the write stage feeds signals back to the decode stage where the registers are typically housed. Note that in the diagram, both the value and the instruction from $MEM-WRI$ are fed back to the decode stage. Although this might seem odd, the instruction is obviously required so that one can extract the target register operand and hence know which register to write the value into: the decode stage cannot know this because it will be operating on a different instruction at the time !

After the write stage has finished, i.e., execution of the instruction is complete, an instruction is said to have been **retired**. Note that with this scheme, strong processor and memory consistency are clearly guaranteed: instructions have to be retired in-order, since no instruction can overtake another one in the pipeline, and there is no re-ordering of memory accesses.

9.2.3 Pipeline Hazards

Due to their complexity and the nature of the tasks they are required to perform, the pipeline stages (as specified) cause hazards as a result of dependencies between the instructions they might deal with. This was not the case, for example, with the pipelined shifter where each stage could work entirely in isolation from the others. As discussed previously, the fact that hazards occur is as a direct result of the data-path design in Figure 9.8: we need some way to identify and resolve hazard conditions otherwise the data-path will execute a given program incorrectly even if it would be correct on a non-pipelined design. In this particular data-path, one can identify several concrete hazard conditions:

Resource Hazards Recall that a resource hazard is where two or more parts of the pipeline need access to the same hardware resource at the same time. Since concurrent access to most resources we have discussed is impossible, this is a problem. For example, we cannot perform an addition and a subtraction at the same time using a single ALU; if we try, it will not compute the correct result.

Data Hazards A RAW type data hazard occurs when an instruction needs an operand which has not yet been generated or not yet reached the write stage. In this case, the processor will try to continue using the wrong operand value and hence the program would be incorrectly executed.

Control Hazards A control hazard occurs where the fetch stage cannot know which address the next instruction should be retrieved from. This is typically because of a conditional branch that has not yet been resolved. If the branch is taken, the processor needs to fetch instructions from the branch target; if it is not taken, the processor just continues fetching from after the branch instruction itself. If the processor just guesses and continues to load instructions from either path and turns out to have made the wrong guess, the pipeline will contain erroneous instructions that could cause the program to execute incorrectly.

We can demonstrate the manifestation of these hazards using a short program fragment. The fragment does not compute anything particularly useful, it just acts to provide a common example for all three cases:

$$
\begin{aligned}
&\ldots \\
&GPR[2] \leftarrow GPR[3] + GPR[4] \\
&GPR[5] \leftarrow GPR[6] - GPR[7] \\
&GPR[8] \leftarrow GPR[2] \cdot GPR[5] \\
&GPR[9] \leftarrow GPR[2] \oplus GPR[5] \\
&\text{if } GPR[8] \neq 0, PC \leftarrow PC + 4 \\
&GPR[8] \leftarrow 1 \\
&GPR[9] \leftarrow GPR[8] \wedge GPR[9] \\
&\ldots
\end{aligned}
$$

All instructions in the program are subject to resource hazards when executed on our example pipelined data-path. This is because both the fetch and execute stages need access to the ALU at the same time; the fetch stage needs to increment the *PC* while the execute stage needs to perform arithmetic in order to execute ALU-dependent instructions. So for example, when the execute stage is dealing with the addition in instruction one using the ALU, the fetch stage will also need the ALU to increment the *PC* after fetching the third, multiply instruction. Note that we have already had to solve this particular hazard in our data-path by having a main ALU in the execute stage and an adder in the fetch stage.

Instruction #2, the multiply, demonstrates a RAW type data hazard. The problem is that when the instruction reaches the decode stage, it tries to fetch the two operands $GPR[2]$ and $GPR[5]$ from the register file. However, the values it fetches will be wrong because the previous two instructions that should produce these results have not yet written them back. Thus, there is a hazard since if execution continues as normal, the result in $GPR[8]$ will be computed incorrectly. This is detailed in Figure 9.9; in step three the diagram shows that as instruction #2 reaches the decode stage instructions #0 and #1, which generate the inputs to instruction #2, are in the memory access and execute stages respectively.

Instruction #4, the conditional branch, demonstrates a control hazard. The problem here is that until instruction #2, the multiply, writes a result into $GPR[8]$ the branch cannot know if that result is zero or not. Hence, it cannot know if the branch should be taken or not and the fetch unit cannot know where to load further instructions from. This is again detailed in Figure 9.9 where, in step five, the fetch unit cannot know where to get instructions from because the branch instruction is still in the decode stage.

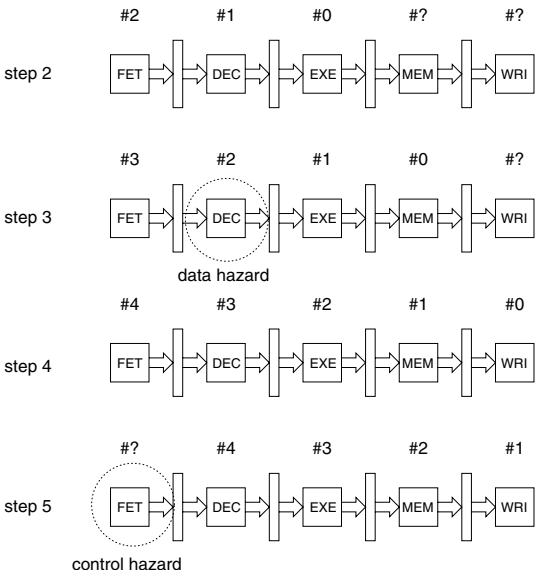


Figure 9.9 A diagram of data and control hazards in the example program.

9.2.4 Stalls and Hazard Resolution

Pipeline hazards present a problem in the sense a given program might not execute as expected. Remember that an ISA is essentially an interface; if we build a pipelined processor that might not execute programs correctly, we are breaking this interface. There are two main approaches to solving this problem:

1. Alter the processor in such a way that all programs will be executed correctly. Such alteration means the processor will be more complex but the programmer will find it easier to use and their programs will be more portable across different implementations of the ISA.
2. Do not alter the processor at all, and place the burden of responsibility for correct execution on the programmer. This means the processor can be simpler, since it does not need to resolve hazards at all, but now the programmer must do more work themselves.

Definition 41. A **stall** is where one or more pipeline of the stages are prevented from advancing so that some activity can finish, for example so that a hazard condition can be resolved or a multi-cycle operation can occur.

Definition 42. A pipeline **bubble** is the "gap" left as the result of a pipeline stage stalling. A NOP instruction is typically inserted into the gap, by either the programmer or the control unit, in order to have it executed correctly.

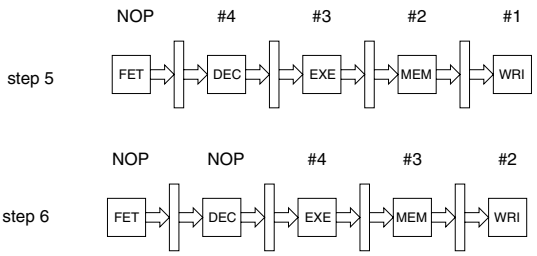


Figure 9.10 Solving a control hazard in the example program using a pipeline bubble.

One normally stalls a pipeline processor while it is waiting for an activity to finish that will take longer than a pipeline cycle. Memory access offers one example of this but complex ALU operations offer another: division (or even multiplication) can often take more than one pipeline cycle and so it is common to see stalls occurring when such multi-cycle operations reach the execution stage. Stages after that which is stalled can be advanced, stages before cannot and have to stall as well; this is because in our simple pipeline the instructions cannot overtake each other. Where we allow stages after the stalled stage to advance, we must fill the gap with something; this something is a pipeline bubble which we represent as a NOP instruction.

Figure 9.10 demonstrates how using a pipeline bubble might work to solve the control hazard presented earlier. Recall that instruction #4 was a conditional branch which could not be resolved until the condition was evaluated; in our case the condition result is fed back from the memory access stage. One can think of this as requiring the fetch unit to stall; stages after it can still advance if we insert pipeline bubbles. So instead of filling the pipeline with invalid instructions, the control unit pads the instruction stream with NOP instructions which have no effect. The program now executes correctly because when the the branch is finally resolved the fetch unit has only injected instructions into the pipeline that do not alter the program correctness; execution can therefore carry on from the branch target without any problem.

Stalling is obviously a problem in terms of performance since it implies some wasted time. Using pipeline bubbles as a solution to hazard resolution demonstrates this: the more useless NOP instructions flowing through the pipeline, the lower the utilisation of resources. Much like the discussion of pipeline utilisation in simple logic circuits, high throughput is only a good thing if we keep the pipeline full of useful instructions. This is especially true if one considers the fact that there are many more instructions that use register operands than those which provoke branches: using pipeline bubbles to solve data hazards would be detrimental to performance as a result. There are several different techniques to reduce the cost associated with resource, data and control hazards; all are valuable because they reduce degradation of performance.

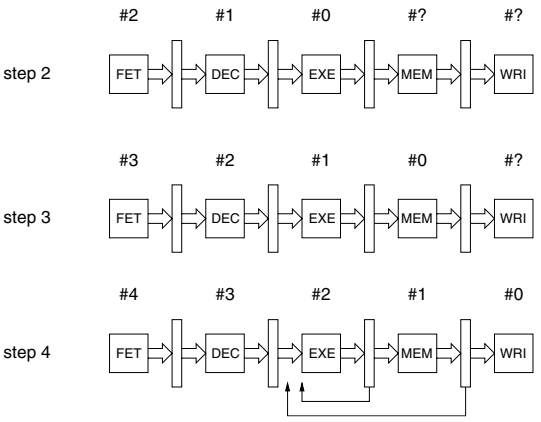


Figure 9.11 Solving a data hazard in the example program using a forwarding.

9.2.4.1 Resolving Resource Hazards

Although we have already addressed the issue of the ALU being required in both the fetch and execute stages of our simple pipeline, resource hazards are more generally solved in one of two ways. The first approach is to do what we have already proposed: simply replicate the common resource so that each stage that needs it has dedicated access. Clearly this is a simple solution in the sense that no real extra management of the resource is required; the drawback is that it will increase the processor size, this is particularly problematic if the replicated resource is large.

The second approach is to time-share a single instance of the resource. This is usually accomplished by splitting a pipeline cycle into phases whereby no two stages can access the same resource in the same phases but otherwise they are free to do what they want. Using the ALU example again, we might say that during the first phase of the pipeline cycle the fetch stage can use the ALU and during the second, disjoint phase the execute stage can have access. This fits well with the requirements of other shared resources as well. For example, the general-purpose registers are effectively shared between the decode and write stages; we might allow reads from the decode stage to happen in the first phase and writes from the write stage to happen in the second phase. Although this does reduce the amount of logic and hence the processor size, a slight caveat is that in order to share the resource one potentially needs to multiplex the inputs and demultiplex the outputs; this adds a small overhead in addition to the added complexity in the control unit.

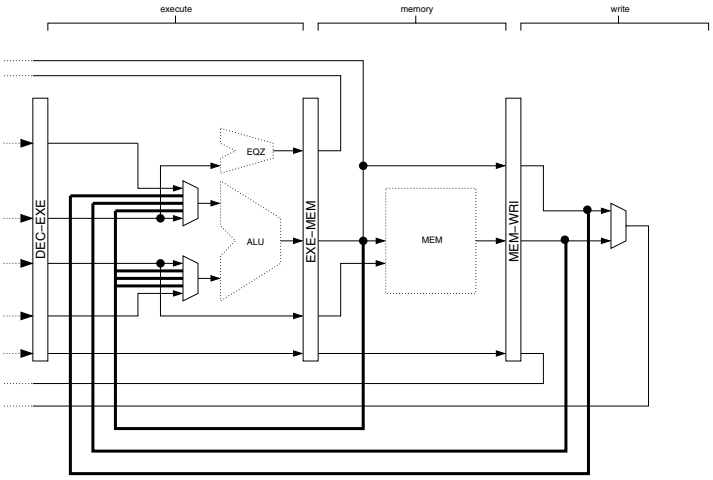


Figure 9.12 Altering the pipelined data-path to include forwarding logic.

9.2.4.2 Resolving Data Hazards Using Forwarding

The key reason we might encounter a RAW type data hazard is that one of the results we want to be in the register file is currently somewhere else within the pipeline; probably it is held as an intermediate result in one of the pipeline registers. So in order to get the right result rather than the erroneous one stored in the register file, all we really need to do is grab the value from a pipeline register rather than use the ones retrieved from the register files. Figure 9.11 demonstrates this strategy: instruction #2 needs the results that should be in $GPR[2]$ and $GPR[5]$ but are not yet because instructions #0 and #1 have not yet reached the write stage. But the results have actually been computed; they are in the pipeline registers so are forwarded instead.

This scheme is fairly easily achieved by extending the number of inputs to the ALU multiplexers and connecting them up to the relevant pipeline registers. The end result looks a little like Figure 9.12. Each multiplexer is operated by the control unit which examines the state of the pipeline at the start of each cycle and sets the multiplexer select signals accordingly; essentially it checks if the source operands of the instruction at the execute phase match the target operand of an instruction at the memory access or write stages. If it does, a hazard is detected and we forward the intermediate result from another stage rather than taking the operands retrieved during the decode stage.

Notice that we have entirely eliminated the performance penalty associated with the detection of this particular data hazard. Although we have increased the amount of logic and complexity in both the pipeline and control unit, the occurrence of a hazard does not require either the pipeline to stall or a bubble to be inserted. As

such, the useful throughput of the pipeline is improved to a level equal with the case where the hazard does not exist.

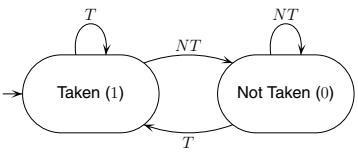
9.2.4.3 Resolving Control Hazards Using Delay Slots

Consider our example from earlier where we stalled the fetch unit and inserted pipeline bubble NOP instructions until the branch condition for instruction #4 was computed. We call this sequence of NOP instructions the **branch delay slot**; one can think of the actual branch execution being "delayed" by this amount. We said that the control unit would insert the NOP instructions for us automatically so that all programs would execute correctly; a different approach would be to let the programmer do this themselves. Although this means more work for the programmer, it allows us to reduce the cost associated with the technique. For example, instruction #3 is independent of the value $GPR[8]$ and as such it does not matter which side of the branch it is executed: it can be moved without altering the program meaning as long as it is actually executed at some point. This ability to reschedule when an instruction is due to be executed allows us to fill the branch delay slot with useful instructions rather than NOP instructions. So we might re-write the program, using horizontal lines to delimit instructions in the branch delay slot, as

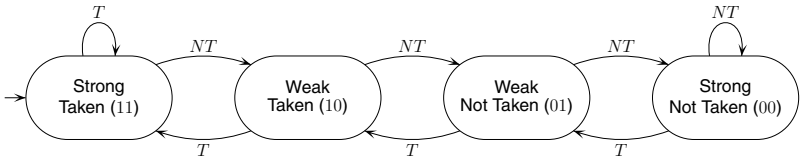
$$
\begin{array}{l}
\ldots \\
GPR[2] \leftarrow GPR[3] + GPR[4] \\
GPR[5] \leftarrow GPR[6] - GPR[7] \\
GPR[8] \leftarrow GPR[2] \cdot GPR[5] \\
GPR[9] \leftarrow GPR[2] \oplus GPR[5] \\
\text{if } GPR[8] \neq 0, PC \leftarrow PC + 4 \\
\hline
NOP \\
NOP \\
\hline
GPR[8] \leftarrow 1 \\
GPR[9] \leftarrow GPR[8] \wedge GPR[9] \\
\ldots
\end{array}
\quad \mapsto \quad
\begin{array}{l}
\ldots \\
GPR[2] \leftarrow GPR[3] + GPR[4] \\
GPR[5] \leftarrow GPR[6] - GPR[7] \\
GPR[8] \leftarrow GPR[2] \cdot GPR[5] \\
\text{if } GPR[8] \neq 0, PC \leftarrow PC + 4 \\
\hline
GPR[9] \leftarrow GPR[2] \oplus GPR[5] \\
NOP \\
\hline
GPR[8] \leftarrow 1 \\
GPR[9] \leftarrow GPR[8] \wedge GPR[9] \\
\ldots
\end{array}
$$

Note that we have had to manually insert NOP instructions to ensure the program executes correctly and that we have left the branch offset the same: in this simple example we will assume the PC value we are incrementing is when the branch is resolved. That is, we assume it takes the value at the end of the delay slot.

Before the change we would have had two NOP instructions in the branch delay slot, now we only have one which is a minor saving but a saving none the less. With more complex programs hopefully it is easy to see that one might be able to move around many more instructions and totally fill the branch delay slot. In this case we totally eliminate the wasted time associated; the trade-off is that our programmer needs to take on more work and correctly construct the delay slot content by hand.



(a) A 1-bit predictor.



(b) A 2-bit predictor.

Figure 9.13 1-bit and 2-bit saturating counter style branch predictors.

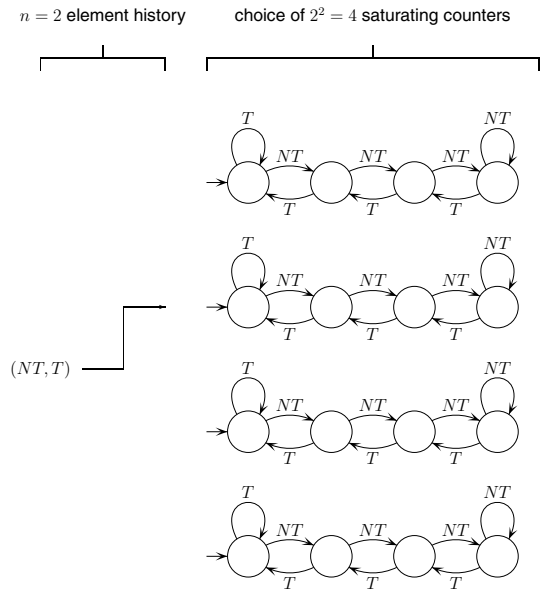


Figure 9.14 A two-level predictor for the case $n = 2$.

9.2.4.4 Resolving Control Hazards Using Speculative Execution

When we discussed control hazards, we stated that we wanted to avoid situations where the pipeline contained instructions we did not want to execute. The opposite approach is loosely called **speculative execution**: we allow such instructions into

the pipeline and clean-up afterwards if it turns out they should not have been there. Essentially we are guessing that they should be there: if our guess is right, then execution can carry on as normal and we have lost nothing, if our guess is wrong (i.e., we **mispredict** the branch) we need to clean-up so the program executes correctly. We call the ratio of mispredictions per executed branch the **misprediction rate**. As long as the misprediction rate is low (i.e., we guess right more often than we guess wrong) performance will be improved over purely inserting useless pipeline bubbles. Implementing a speculative execution scheme requires three mechanisms: one to control the pipeline correctly; one that performs **branch prediction** to guess if a branch will be taken or not; and one that performs **branch target prediction** to guess where a taken branch will redirect control-flow to.

No matter how good the branch prediction technique is, the processor will sometimes make the wrong guess. In order to clean-up a situation where there are erroneous instructions in the pipeline we need to delete them. One option is to **flush** the entire pipeline, emptying it of instructions, and starting again. However we can be slightly more clever than this and simply replace them with NOP instructions; we say that the erroneous instructions are **squashed**. Squashing the erroneous instructions is slightly harder from a control perspective but helps to avoid a full pipeline flush which will typically be more expensive.

There are numerous approaches to branch prediction; the subject remains an active research topic because even small improvements in probability of correct guesses can have a significant impact on performance. The approaches can be coarsely grouped into two classes which we expand on below. By static prediction we basically mean prediction which occurs at compile-time, although more precisely one should make a distinction between **fixed prediction** and **static prediction**:

Fixed Prediction One of the most simple approaches would be to assume a single, fixed view of whether branches are taken or not; this is **fixed prediction**. Such an approach clearly cannot offer much of an advantage, however, since the prediction made is totally independent of the program structure or the data it is processing, both of which will have an influence. As such, additional heuristics or rules of thumb are often used to improve matters. For example, one might guess that backward branches are always taken while forward branches are not taken. This rule of thumb models the way loops typically work: the branch back to the beginning of the loop only fails once [3, 61].

Static Prediction Like many instruction sets, the full MIPS32 ISA includes a number of branch instructions that operate as normal but additionally hint to the processor that the branch will be taken or not taken. The idea is that a compiler or programmer can analyse a program at compile-time, work out the most likely path and pass this information on to be used by the processor; this is **static prediction**. For example, the MIPS32 `bltz` branch also comes in an alternate guise called `bltzl` which signals that it is likely that the branch will be taken.

In contrast to static approaches, **dynamic prediction** typically maintains a historical record of branch behaviour at run-time. This information maps branch addresses

to information which is used, according to some scheme, to predict whether that particular branch will be taken or not:

Saturating Counter One scheme formulates the prediction task as an FSM [40]. Imagine the processor maintains a small cache-like structure called a **Branch Prediction Buffer (BPB)** which is used to map a branch to a 1-bit history which represents a current state in the FSM; each FSM state is labelled to show whether we should predict the branch will be taken or not. Figure 9.13a shows the FSM. The history, and hence state, is updated with the actual outcome each time we execute a branch: if it is taken (denoted T) or not taken (denoted NT) we make the appropriate transition in the FSM and hence update our next prediction.

Of course, a 1-bit history has limited capability so one can generalise the idea to an n-bit history. For example, Figure 9.13b shows the FSM for a 2-bit history which allows four FSM states in total: in the two left-hand states we predict the branch is taken but the most left state is a more confident prediction. This essentially gives a "second chance" to the branch, for example, if it is almost always taken but not taken just once we still predict correctly with a 2-bit history, yet make a mistake with a 1-bit history. This scheme is sometimes called a **saturating counter**-based branch predictor since one can view the value representing the history as a counter which is incremented or decremented based on branch behaviour and which saturates (i.e., stops) at either extreme.

Two-Level Predictor Consider a 2-bit saturating counter-based predictor and a branch that alternates between taken and not taken, i.e., the i-th time one executes the branch it is taken if $i \equiv 0 \pmod 2$ and not taken otherwise. If the predictor is initialised in the "strongly taken" state then it will alternate between the "strongly taken" and "weakly taken" states and hence only predict correctly half of the time.

The so-called **two-level predictor** [69] is one approach to resolving the problem of such patterns. The idea is to split each entry in the BPB into two parts: one buffer which tracks the exact history of the associated branch over the last n executions, and one which holds 2^n saturating counter predictors that correspond to each possibility. This approach, as roughly shown in Figure 9.14 for just one BPB entry, clearly adds some overhead ! However, at least with the example above, it solves the problem. Consider the case of $n = 2$ so that each entry in the BPB holds the exact history of a given branch over the last two executions. There are $2^2 = 4$ saturating counters, one each for the cases (NT,NT), (NT,T), (T,NT) and (T,T). Each of the counters will "learn" what the correct prediction is in each case and, crucially, not interfere with each other; the important cases for the example above are (NT,T) and (T,NT) where the branch alternates between taken and not taken. The history will select the appropriate counter which will always predict correctly.

In the case where branch prediction indicates a branch will be taken, the processor needs to know where the branch target is so it can set the *PC* value. A common method is to maintain another small cache-like structure called a **Branch Target Buffer (BTB)** which maps branch instructions to their predicted target. When ex-

ecution reaches a branch which is predicted to be taken, the processor checks the BTB. If the branch instruction is present, then instructions are speculatively fetched from the predicted target. Of course, since the BTB size is limited a branch might not be present when we need to make our guess; if it is not, we make an uninformed, static guess. Either way, once we know the real outcome we can update the predicted branch target to improve the guess next time the branch is encountered.

9.3 Superscalar Processors

9.3.1 Basic Concept

In terms of design and operation, one can identify a number of characteristics in the pipelined processor we have already described. Firstly, it represents only a basic approach to exploiting ILP: the processor parallelises with respect to stages of sequential execution. Put more simply, the pipeline does not have multiple PEs which allow the various types of parallelism demonstrated at the beginning of the chapter; it cannot for example have two instructions in the execute stage at once. Being very precise one might not view pipelining as parallelism per-se since each stage in the pipeline is performing a different task. Secondly, the execution of instructions is in-order. That is, if instruction X enters the pipeline before instruction Y, then it always retires before Y. Finally, pipelining only really exploits parallelism in straight-line instruction sequences; as soon as a branch occurs, the pipeline fails in some sense. These features have important implications in terms of the limits on performance improvement such designs can yield. For example, if the instruction X stalls the pipeline, Y cannot proceed even if there is no dependency between them. Further, if X uses a different element of the ALU than Y, for example it performs floating point rather than integer arithmetic, then the hardware used to execute Y will idle while X is being dealt with and vice versa.

Put very simply, the idea of a **superscalar** processor is to have a single processor **front-end** which is tasked with preparing instructions for execution and a **back-end** which consists of several execution pipelines. Viewing each execution pipeline as a PE of sorts, the front-end and back-end are responsible (in combination) for producing two major improvements in relation to the single-issue pipelined processor. Firstly, the processor is able to cope with several in-flight instructions at once; **dynamic scheduling** places instructions into the relevant execution pipeline. Secondly, as a result of dynamic scheduling, the processor can potentially perform **multiple-issue** by placing more than one instruction into an execution pipeline during the same clock cycle. The combination of these improvements gives superscalar performance.

Figure 9.15 attempts to describe the structure of a superscalar processor in a way that offers a stepping stone from the single-issue pipelined processor. As such, one can view the diagram as retaining the fetch and decode stages in some form.

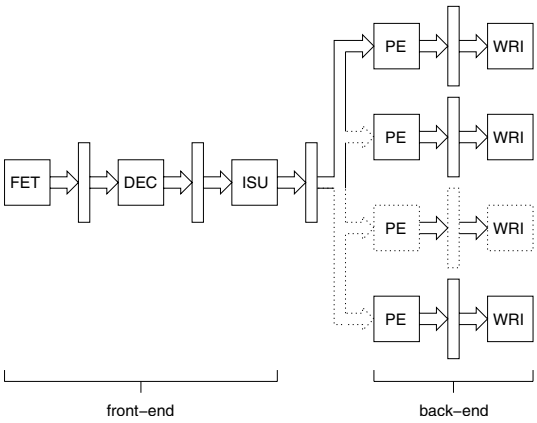


Figure 9.15 A conceptual diagram of a superscalar processor; the part-dashed path between *ISU* and the PEs is supposed to show that a given instruction is passed to one PE only.

It includes an issue stage denoted *ISU* which notionally maintains an **issue buffer** of instructions ready to be scheduled. Finally it replicates the other stages to form execution pipelines specialised to particular tasks. For example, one might have an arithmetic pipeline to deal with integer arithmetic and a load/store pipeline to deal with memory accesses. As such, if a given program contains an arithmetic instruction *X* and a load instruction *Y* which have no dependencies on each other, we might expect the processor to feed the arithmetic pipeline with *X* and the load/store pipeline with *Y* thereby executing *X* and *Y* in parallel.

The design of superscalar processors is difficult; this subject probably represents the most demanding material covered in this book. Our aim is to describe the basic concepts (rather than many details) at an intentionally high level; we use two designs to step from the original single-issue pipelined processor to a multiple-issue superscalar processor. First we introduce the concept of a **scoreboard** and how this basic design can perform the dynamic scheduling aspect. Then we improve on the design, by replacing the scoreboard with **reservation stations** and adding a **Re-Order Buffer (ROB)**, to realise the multiple-issue aspect. Following these steps we briefly comment on a number of embellishments which can be used to further improve performance.

9.3.2 Step 1: Scoreboard-based Design

The name **scoreboard** originally stems from the CDC-6600, a mainframe computer from Control Data Corporation deployed in the mid 1960s. The scoreboard is a set of tables which keeps track of the resources used by multiple in-flight instructions

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$				
$GPR[4] \leftarrow MEM[4]$				
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$				
$GPR[6] \leftarrow GPR[3] + GPR[4]$				
$GPR[7] \leftarrow GPR[5] - GPR[4]$				
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	0							1	1
MUL	0							1	1

	GPR[0]	*GPR*[1]	*GPR*[2]	*GPR*[3]	*GPR*[4]	*GPR*[5]	*GPR*[6]	*GPR*[7]	...	*GPR*[31]
PE									...	

Cycle t_0 starts by initialising the processor and the associated scoreboard tables.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1			
$GPR[4] \leftarrow MEM[4]$				
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$				
$GPR[6] \leftarrow GPR[3] + GPR[4]$				
$GPR[7] \leftarrow GPR[5] - GPR[4]$				
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	1	#0	3					1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	0							1	1
MUL	0							1	1

	GPR[0]	*GPR*[1]	*GPR*[2]	*GPR*[3]	*GPR*[4]	*GPR*[5]	*GPR*[6]	*GPR*[7]	...	*GPR*[31]
PE				*LSU*					...	

Cycle t_1 issues instruction #0.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$				
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$				
$GPR[6] \leftarrow GPR[3] + GPR[4]$				
$GPR[7] \leftarrow GPR[5] - GPR[4]$				
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	0							1	1
MUL	0							1	1

	GPR[0]	*GPR*[1]	*GPR*[2]	*GPR*[3]	*GPR*[4]	*GPR*[5]	*GPR*[6]	*GPR*[7]	...	*GPR*[31]
PE									...	

Cycles $t_2 \ldots t_8$ see instruction #0 execute and retire.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9			
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$				
$GPR[6] \leftarrow GPR[3] + GPR[4]$				
$GPR[7] \leftarrow GPR[5] - GPR[4]$				
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	1	#1	4					1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	0							1	1
MUL	0							1	1

	GPR[0]	*GPR*[1]	*GPR*[2]	*GPR*[3]	*GPR*[4]	*GPR*[5]	*GPR*[6]	*GPR*[7]	...	*GPR*[31]
PE					*LSU*				...	

Cycle t_9 issues instruction #1.

Figure 9.16 Execution of a simple program on a scoreboarded superscalar processor.

	ISU	RD	EXE	RET
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}		
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}			
$GPR[6] \leftarrow GPR[3] + GPR[4]$				
$GPR[7] \leftarrow GPR[5] - GPR[4]$				
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	O	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	1	#1	4					1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	0							1	1
MUL	1	#2	5	2	4		LSU	1	0

	GPR[0]	GPR[1]	GPR[2]	GPR[3]	GPR[4]	GPR[5]	GPR[6]	GPR[7]	...	GPR[31]
PE					LSU	MUL			...	

Cycle t_{10} issues instruction #2.

	ISU	RD	EXE	RET
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}			
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}			
$GPR[7] \leftarrow GPR[5] - GPR[4]$				
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	O	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	1	#1	4					1	1
BU	0							1	1
ALU_0	1	#3	6	3	4		LSU	1	0
ALU_1	0							1	1
MUL	1	#2	5	2	4		LSU	1	0

	GPR[0]	GPR[1]	GPR[2]	GPR[3]	GPR[4]	GPR[5]	GPR[6]	GPR[7]	...	GPR[31]
PE					LSU	MUL	ALU_0		...	

Cycle t_{11} issues instruction #3 while instruction #1 progresses into execution.

	ISU	RD	EXE	RET
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}			
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}			
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	O	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	1	#1	4					1	1
BU	0							1	1
ALU_0	1	#3	6	3	4		LSU	1	0
ALU_1	1	#4	7	5	4	MUL	LSU	0	0
MUL	1	#2	5	2	4		LSU	1	0

	GPR[0]	GPR[1]	GPR[2]	GPR[3]	GPR[4]	GPR[5]	GPR[6]	GPR[7]	...	GPR[31]
PE					LSU	MUL	ALU_0	ALU_1	...	

Cycle t_{12} issues instruction #4; instructions #2, #3 and #4 are now blocked waiting for instruction #1.

	ISU	RD	EXE	RET
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}			
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}			
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	O	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	1	#3	6	3	4			1	1
ALU_1	1	#4	7	5	4	MUL		0	1
MUL	1	#2	5	2	4			1	1

	GPR[0]	GPR[1]	GPR[2]	GPR[3]	GPR[4]	GPR[5]	GPR[6]	GPR[7]	...	GPR[31]
PE						MUL	ALU_0	ALU_1	...	

Cycles $t_{13} \ldots t_{16}$ see instruction #1 execute and retire.

Figure 9.17 Execution of a simple program on a scoreboarded superscalar processor.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}		
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}		
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	1	#3	6	3	4			1	1
ALU_1	1	#4	7	5	4	*MUL*		0	1
MUL	1	#2	5	2	4			1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE						*MUL*	ALU_0	ALU_1	...	

Cycle t_{17} shows instruction #1 retire, allowing instructions #2 and #3 to progress by reading their operands.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	1	#3	6	3	4			1	1
ALU_1	1	#4	7	5	4	*MUL*		0	1
MUL	1	#2	5	2	4			1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE						*MUL*	ALU_0	ALU_1	...	

Cycle t_{18} sees instructions #2 and #3 progress into execution.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	t_{21}
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$				

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	1	#4	7	5	4	*MUL*		0	1
MUL	1	#2	5	2	4			1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE						*MUL*		ALU_1	...	

Cycles $t_{19} \ldots t_{21}$ see instruction #3 execute and retire.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	t_{21}
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$	t_{22}			

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	1	#5	4	6	4			1	1
ALU_1	1	#4	7	5	4	*MUL*		0	1
MUL	1	#2	5	2	4			1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE					ALU_0	*MUL*		ALU_1	...	

Cycle t_{22} issues instruction #5; instruction #4 is blocked waiting for the result of instruction #2.

Figure 9.18 Execution of a simple program on a scoreboarded superscalar processor.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	t_{21}
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$	t_{22}	t_{23}	t_{24}	t_{25}

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	1	#5	4	6	4			1	1
ALU_1	1	#4	7	5	4	*MUL*		0	1
MUL	1	#2	5	2	4			1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE					ALU_0	*MUL*		ALU_1	...	

Cycles $t_{23} \ldots t_{25}$ see instruction #5 finish execution but then get blocked from retiring because of instructions #4.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	t_{27}
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	t_{21}
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}			
$GPR[4] \leftarrow GPR[6] + GPR[4]$	t_{22}	t_{23}	t_{24}	t_{25}

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	1	#5	4	6	4			1	1
ALU_1	1	#4	7	5	4			1	1
MUL	0							1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE					ALU_0			ALU_1	...	

Cycles $t_{26} \ldots t_{27}$ see instruction #2 execute and retire.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	t_{27}
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	t_{21}
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}	t_{28}		
$GPR[4] \leftarrow GPR[6] + GPR[4]$	t_{22}	t_{23}	t_{24}	t_{25}

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	1	#4	7	5	4			1	1
MUL	0							1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE								ALU_1	...	

Cycle t_{28} sees instruction #5 retire.

	ISU	*RD*	*EXE*	*RET*
$GPR[3] \leftarrow MEM[0]$	t_1	t_2	t_3	t_8
$GPR[4] \leftarrow MEM[4]$	t_9	t_{10}	t_{11}	t_{16}
$GPR[5] \leftarrow GPR[2] \cdot GPR[4]$	t_{10}	t_{17}	t_{18}	t_{27}
$GPR[6] \leftarrow GPR[3] + GPR[4]$	t_{11}	t_{17}	t_{18}	t_{21}
$GPR[7] \leftarrow GPR[5] - GPR[4]$	t_{12}	t_{28}	t_{29}	t_{30}
$GPR[4] \leftarrow GPR[6] + GPR[4]$	t_{22}	t_{23}	t_{24}	t_{25}

	B	*O*	F_a	F_b	F_c	Q_b	Q_c	R_b	R_c
LSU	0							1	1
BU	0							1	1
ALU_0	0							1	1
ALU_1	0							1	1
MUL	0							1	1

	$GPR[0]$	$GPR[1]$	$GPR[2]$	$GPR[3]$	$GPR[4]$	$GPR[5]$	$GPR[6]$	$GPR[7]$	...	$GPR[31]$
PE									...	

Cycles $t_{29} \ldots t_{30}$ see instruction #4 retire.

Figure 9.19 Execution of a simple program on a scoreboarded superscalar processor.

as they progress through execution. The information in the scoreboard can thus be used to exert *centralised* control over instruction scheduling so that, for example, an instruction cannot be issued before the associated operands are available. The CDC-6600 employed this system to great effect, keeping ten parallel PEs (mainly floating point execution units) supplied with instructions.

9.3.2.1 Scoreboard

Roughly speaking, there are three components in a scoreboard:

Definition 43. The scoreboard contains an **instruction table** called IT which records the status of each in-flight instruction, i.e., whether the instruction is being issued, reading operands, in execution or writing results. The number of entries in the table should be equal to the largest possible number of in-flight instructions; we use $IT[j]$ to refer to the j-th such entry and make the simplifying assumption that the table is always large enough to cope with our examples.

Definition 44. The scoreboard contains an **execution table** called ET which records the status of each PE. There is one entry in the table for each of the PEs; we use $ET[j]$ to refer to the j-th such entry which contains the following fields:

1. $ET[j]_B$, a flag which is conditionally set depending on whether the j-th PE is in a busy or idle state.
2. $ET[j]_O$, a field which details the operation the j-th PE will perform.
3. $ET[j]_{F_a}$, the target register of the instruction being dealt with by the j-th PE, or blank if the PE is idle; $ET[j]_{F_b}$ and $ET[j]_{F_c}$, the source registers of the instruction being dealt with by the j-th PE, or blank if the PE is idle.
4. $ET[j]_{Q_b}$ and $ET[j]_{Q_c}$, the PE numbers which will produce the source operands in $ET[j]_{F_b}$ and $ET[j]_{F_c}$
5. $ET[j]_{R_b}$ and $PR[j]_{R_c}$, flags which are conditionally set depending on whether $ET[j]_{F_b}$ and $ET[j]_{F_c}$ contain operands that are ready for use, or whether the operands are yet to be produced by the PEs specified by $ET[j]_{Q_b}$ and $ET[j]_{Q_c}$.

Definition 45. The scoreboard contains a **register table**, which in reality might be merged with the register file, containing details for each register that show which PE is currently engaged in an operation which will write into it. There is an entry for each register in the general-purpose register file; we use $RT[j]$ to refer to the j-th such entry.

9.3.2.2 Control Strategy

Using components detailed in Figure 9.20, execution of instructions is performed by the following control strategy:

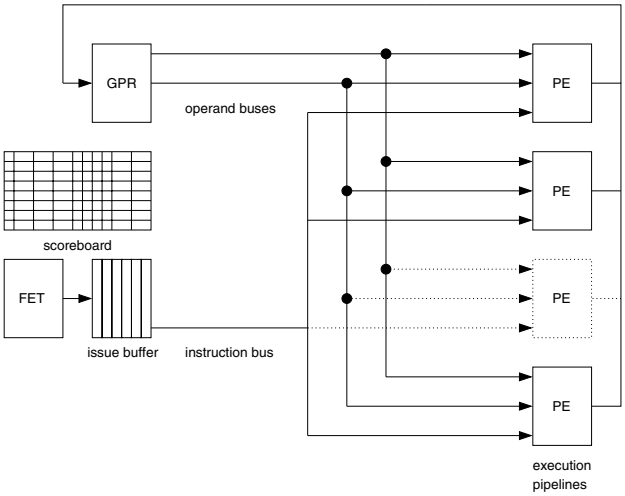


Figure 9.20 Step 1: a scoreboard-based superscalar data-path.

Fetch (FET) **and Decode** (DEC) Instructions are fetched from memory as normal and decoded (so, for example, it is clear which source and target registers are used) into the **issue buffer**; the scoreboard tables are updated to include the instruction.

Note that unlike the single-issue pipeline described previously, the *DEC* stage does not fetch register operands from the register file; this task is deferred until later.

Issue (ISU) An instruction X in the issue buffer is stalled in the ISU stage until

- all instructions Y that target the same register as X retire, and
- some PE capable of processing X is idle.

These conditions cope with the potential for WAW and resource hazards respectively; when they are satisfied, the instruction is allowed to progress to the *RD* stage and the scoreboard tables are updated.

Read (RD) Before actually being executed, an instruction X stalls again in the *RD* stage until the required source operands are available; this copes with the potential for RAW hazards. When the operands are available, they are read and the instruction is passed to the associated PE.

Execute (EXE) An instruction X then starts to be executed by the associated PE. As a result of passing through different PEs, different instructions might take different lengths of time to finish execution: a later instruction Y in program order might finish execution before X does.

Retire (RET) An instruction X is stalled again before writing-back result so all instructions Y, that use the same register as a source operand that X produces

as a target operand, finish the read stage. This copes with the potential for WAR hazards. At this point, X writes any results to target registers and the instruction retires before being deleted from the scoreboard tables.

This strategy has some characteristics and related terminology which need to be highlighted. Firstly, note that by design there are three simplifications which are less than ideal. Firstly, although the processor can perform dynamic scheduling to keep multiple PEs busy at once, it cannot issue more than one instruction per-cycle. Secondly, there is no processor consistency: there is nothing to say an instruction X that passes through a low-latency PE might finish execution before another instruction Y that passes through a PE with higher latency, even if X comes after Y in program order. Thirdly, we have glossed over the execution of branches: as such one should assume there is no speculative execution, once a branch is fetched into the issue buffer it blocks all subsequent instructions until resolved. Finally, we cannot cope with the fact that a given PE might be sub-pipelined: only one instruction can reserve the PE as a resource at a time. This could potentially be resolved by adding each sub-pipeline stage as a pseudo-PE in the scoreboard table, but for now we ignore this.

Other aspects of the strategy are choices that we will expand on later. In particular, the strategy uses what is called **in-order issue**. This means that instructions can only be issued to PEs in the same order they appear in the issue buffer; if instruction X is before instruction Y in the buffer, X is always issued before Y even if X is blocked and Y could be issued. Put another way, this means the issue buffer operates as a **First-In First-Out (FIFO)** structure.

9.3.2.3 Example Execution

Since the main aim of the control strategy above is to schedule instructions into PEs dynamically, the actual behaviour is highly dependent on the program a processor is executing. In that sense, it is difficult for a single example to capture all the cases which could occur. On the other hand, without an example it is difficult to get a grasp on how the control strategy, and hence the processor, behaves. We try to compromise between these two factors by using the program fragment

$$
\begin{aligned}
&\ldots\\
&GPR[3] \leftarrow MEM[0]\\
&GPR[4] \leftarrow MEM[4]\\
&GPR[5] \leftarrow GPR[2] \cdot GPR[4]\\
&GPR[6] \leftarrow GPR[3] + GPR[4]\\
&GPR[7] \leftarrow GPR[5] - GPR[4]\\
&GPR[4] \leftarrow GPR[6] + GPR[4]\\
&\ldots
\end{aligned}
$$

The idea is to consider this program executing on a scoreboarded superscalar processor that contains five different PEs: LSU, a **Load/Store Unit (LSU)** which is tasked

with execution of memory accesses; *BU*, a **Branch Unit (BU)** which is responsible for executing branch instructions; ALU_0 and ALU_1, two integer ALUs which execute all single-cycle integer instructions (e.g., addition and subtraction); and a dedicated unit for multiplication called *MUL*. We will assume that the processor is only able to perform single-issue, i.e., that only one instruction can be issued by the front-end into the back-end per-cycle. We already noted that our control strategy cannot currently cope with any PE being sub-pipelined. However, because of their different behaviour, the PEs have different latencies associated with them: *LSU* takes five cycles to perform a memory access; *BU* takes three cycles to process a branch; ALU_0 and ALU_1 take three cycles to perform integer arithmetic; and *MUL* takes nine cycles to multiply numbers together. As a result, although the processor is single-issue it is still possible to have multiple instructions being executed in parallel.

Figure 9.16 through to Figure 9.19 detail (at some length) program execution. In each part, instead of writing simply true or false in the *IT* table when an instruction has completed or not completed a given stage, we write the cycle in which this event occurs: this gives a more instructive way to track execution as it progresses. Where no interesting events occur within a given period, we omit the associated diagram for brevity.

From cycles t_0 to t_9, nothing too interesting happens: instructions #0 and #1, the two loads, start to execute. The second one is blocked from being issued by the first one because of the resource hazard (i.e., there is only one *LSU*). The first load starts at t_3 and eventually retires five cycles later at t_8 which allows the second load to be issued at t_9. In cycle t_{10} more interesting things start to happen. In particular, we can issue instruction #2, the multiply, into *MUL* while the load is being processed by *LSU*. Note that cycle t_{10} represents a point where we could perform multiple-issue if able to do so, i.e., issue instruction #2 and instruction #3 in the same cycle. At cycle t_{11} we see that instruction #1 is being executed but instruction #2 is blocked at the read stage because it is waiting for the value in $GPR[4]$ to become available. That is, the instruction is blocked from progress to avoid the RAW hazard associated with the dependency between the instructions; one can see in the *RT* table that the *LSU* will produce the value in $GPR[4]$ and in the *ET* table that the $ET[4]_{R_c}$ flag, i.e., the one for the *MUL* PE, is zero meaning that the second operand is not available. Also in cycle t_{11} we can issue instruction #3 into ALU_0. In cycle t_{12} we can issue instruction #4 into the ALU_1 PE. Note that we now have three instructions waiting for *LSU* to finish executing the second load instruction, and that instruction #4 is additionally waiting for *MUL* to finish execution of the multiply. Finally, in cycle t_{16} the load retires and the value in $GPR[4]$ is ready for use; this allows instructions #2 and #3 to progress through the read stage in t_{17}. Note that instruction #4 is still blocked because it is waiting for the result of multiplication. In cycle t_{18} both instructions #2 and #3 begin execution; instruction #3 finishes execution three cycles later in cycle t_{21} but instruction #2, which has a much higher latency, is still being dealt with. After instruction #3 retires, ALU_0 becomes free which means we can issue instruction #5 to it in cycle t_{22}: before this, instruction #5 was blocked as a result of the associated resource hazard.

Cycles t_{23} to t_{25} detail the progress of instruction #5; note that instruction #4 is blocked waiting for the result of instruction #2 which is still being executed. Cycle t_{25} is interesting because one might expect instruction #5 to retire. However, it is blocked from doing so because it would write to $GPR[4]$, the same register that instruction #4 is yet to read from: to avoid the WAR hazard that results from the associated dependency, instruction #5 is essentially blocked from retiring until instruction #4 has finished the read stage. At cycle t_{27} the multiplication finally finishes and instruction #2 retires. This allows instruction #4 to read the required operands in cycle t_{27} and, in turn, for instruction #5 to retire by writing-back the result it had been hanging on to. Finally, the program fragment finishes execution in cycle t_{30} when instruction #4 retires.

9.3.3 Step 2: Reservation Station-based Design

By examining the initial design above, particularly looking at the example execution, one can identify a few drawbacks. Firstly, there are clearly some points which we knowingly avoided (for example multiple-issue, pipelined PEs); of course one can improve the design in these respects. Furthermore, one might conceptually view scoreboarding as unattractive because of scalability. One argument against centralised control could be that it is harder to add more PEs to the system since the scoreboard will grow very large.

Secondly, there are some more subtle features. The use of an in-order, blocking single-issue strategy means that potentially an instruction can block the issue of another where there is no requirement to do so. For example, consider some non-dependent instructions X and Y where X occurs before Y in program order. If there is no free PE to accept instruction X, instruction Y is blocked from being issued even if there is a free PE to accept it. Furthermore, there are other stages of execution where instructions can be stalled where there is no need. Again consider an example of instructions X and Y which this time write to the same register, i.e., imply a WAW hazard and hence potentially stall during the issue stage. We have already seen that this form of dependency is in some sense a fake: if the programmer (or processor) could re-write the program somehow so that Y used a different register, this potential to stall is removed. Clearly these features are undesirable since they degrade performance.

In the mid to late 1960s Robert Tomasulo, then a researcher at IBM, developed what is now a fundamental algorithm [62] for improved dynamic scheduling. The algorithm was developed to manage floating-point PEs within the IBM 360/91 computer. Roughly speaking, there are three main alterations to the initial design from above:

1. The centralised scoreboard is (mostly) removed, with control information delegated to a number of **reservation stations**. Each reservation station is similar to the combination of an issue buffer and scoreboard tables. The basic idea is to

buffer or **shelve** instructions, controlling them in a more decentralised way, until they are ready to be **dispatched** into the corresponding PE.
2. A so-called **Common Data Bus (CDB)** is implemented as the path by which instructions write results to the register file. The main innovation here is a forwarding mechanism whereby if an instruction buffered within a reservation station requires a value being written into the register file, it can collect the value "early" rather than waiting until it can be subsequently read.
3. A **Re-Order Buffer (ROB)** structure is added to perform three duties:

   a. Resolve the lack of processor consistency by ensuring instructions retire in-order even if they reach the retire stage out-of-order.
   b. Enable a mechanism called **register renaming**. Conceptually, one can see this as splitting the definition of registers into two: an **architectural register** is a register specified as part of an instruction and as prescribed by the ISA, a **physical register** is the actual hardware register that a reference to a specific architectural register is **backed** by.
   Note that we can have more physical registers within the processor than architectural registers specified in the ISA. The technique is register renaming, very roughly, means the processor mapping between the two sets dynamically so as to remove WAR and WAW style hazards.
   c. Enable speculative execution to occur so that the high cost of branches (due to their blocking every other instruction until resolved in the initial design) can be reduced.

9.3.3.1 Reservation Stations and Common Data Bus (CDB)

Even with reservation stations, we retain two aspects of the scoreboard-based design, namely the IT and RT tables which record instruction status for each in-flight instruction, and the reservation station slot (previously PE) which will generate each register. Each reservation station records similar but subtly different information to the scoreboard table ET. The idea is that each reservation station is issued instructions; each acts as a buffer where such instructions wait until their operands are available and they can be dispatched into an associated PE.

Definition 46. The i-th reservation station RS_i has a number of slots, i.e., the table RS_i has a number of rows; we use $RS_i[j]$ to refer to the j-th such slot in the i-th reservation station which contains the following fields:

1. $RS_i[j]_B$, a flag which is conditionally set depending on whether the PE associated to the i-th reservation station is in a busy or idle state.
2. $RS_i[j]_O$, a field which details the operation the PE associated to the i-th reservation station will perform for the instruction in slot j.
3. $RS_i[j]_{V_b}$ and $RS_i[j]_{V_c}$, the value of the source operands (previously the register number) used by the instruction in slot j within the i-th reservation station, or is blank if the value is not yet available.

4. $RS_i[j]_{Q_b}$ and $RS_i[j]_{Q_c}$, the reservation station slot which will produce the source operands in $RS_i[j]_{V_b}$ and $RS_i[j]_{V_c}$ or is blank if those operands are already available.

In a sense, one can view the reservation stations simply like a distributed scoreboard except that it is *value* based rather than *address* based: it stores the value of operands rather than the register number where the operand is located. This implicitly means the reservation stations manage the problem of RAW hazards in that an instruction cannot be dispatched from a slot before the operand values are filled.

If an operand value already exists within the register file, it is read during issue of the instruction into the reservation station. If not, i.e., there is some other instruction being executed that will generate the value as a result, then normally the instruction would stall until the value is available: this means waiting for the associated read from the register file. The CDB, which passes results from the PEs to be written into the register file, provides a significant advantage over this situation by allowing a form of forwarding. Imagine each reservation station sits monitoring the CDB. When a value passes along it which is required as an operand value in some slot, the value is copied directly from the CDB into that slot; this in turn potentially enables dispatch to occur at once rather than having to wait for the operand to be written to and then read from the register file.

Issue of instructions into the reservation stations is still in-order. However, imagine there are two instructions X and Y in a reservation station and they both become ready for dispatch at the same time (maybe one value on the CDB enables two instructions at the same time). This implies the need to choose which of them to dispatch. The reservation station could dispatch the instruction which is first in program order, performing the analogy of in-order issue that we call **in-order dispatch**. However, given X and Y are both free from dependency there is no real reason why such a strategy is necessary: the reservation could perform **out-of-order dispatch** by selecting an instruction more arbitrarily.

Note that where the processor can dispatch more than one instruction per-cycle, the CDB must be able to support the fact that multiple instructions may want to write results at the same time. If this were not the case, dispatch from the front-end would soon be prevented as instructions form a traffic jam in the back-end.

9.3.3.2 Re-Order Buffer (ROB)

The concept of a **Re-Order Buffer (ROB)** is the somewhat canonical way to ensure processor and memory consistency which has, up until now, been lacking.

Definition 47. The ROB is basically a circular buffer; it has a number of slots, where we denote the i-th such slot as $ROB[i]$, and a head and a tail pointer that we denote ROB_{head} and ROB_{tail}.

When instructions are ready to be issued, they are allocated an entry in the ROB at the tail pointer. When instructions are ready to retire, they can only do so once

all previous instructions resident in the ROB have retired; essentially this means instructions are removed from the ROB at the head pointer. Only when an instruction is completed by removing it from the ROB is it allowed to write any results to the architectural registers and/or memory. Essentially this enforces **in-order completion**: even if instructions are dispatched out-of-order or overtake each other within PEs of different latency, they still retire in-order so that consistency is maintained. Note that where the processor can issue more than one instruction per-cycle, the ROB needs to be able to retire more than one instruction per-cycle otherwise the same traffic jam occurs as described above.

The second benefit of the ROB is that it gives a fairly simple solution to implementation of **register renaming**. In concept, register renaming is fairly easy. Imagine there are n architectural registers addressable by the programmer via the ISA, and the i-th architectural register is $GPR[i]$. However, the abstractly described architectural registers are implemented, or **backed**, by physical registers within the processor. Imagine the physical register file is some larger collection, say of size m registers, where the i-th physical register is $PRF[i]$. With such a design a **rename table**, say RNT, is maintained to map architectural registers to physical registers. Every time an instruction uses an architectural register as a source operand, we use RNT to map the address to one of the physical registers. Every time an instruction uses an architectural register as a target operand, we allocate an unused physical register use that as the target and update RNT to remap subsequent register addresses. Consider a case where there are 32 architectural registers and 64 physical registers and where the RNT is initialised to

$$\begin{aligned} RNT[0] &= 0 \\ RNT[1] &= 1 \\ RNT[2] &= 2 \\ RNT[3] &= 3 \\ RNT[4] &= 4 \\ RNT[5] &= \perp \\ &\vdots \\ RNT[31] &= \perp \end{aligned}$$

Here the $\perp$ symbol is used where no physical register has yet been allocated to the corresponding architectural register. At the start of the chapter we described the problem of so-called "fake" dependencies; these can be dynamically resolved by a processor using register renaming:

$$\begin{array}{l} \ldots \\ GPR[2] \leftarrow GPR[3] + GPR[4] \\ GPR[5] \leftarrow GPR[2] + GPR[2] \\ GPR[2] \leftarrow GPR[3] - GPR[4] \\ \ldots \end{array} \quad \mapsto \quad \begin{array}{l} \ldots \\ PRF[5] \leftarrow PRF[3] + PRF[4] \\ PRF[6] \leftarrow PRF[5] + PRF[5] \\ PRF[7] \leftarrow PRF[3] + PRF[4] \\ \ldots \end{array}$$

The register addresses of source operands in instruction #0 are mapped using *RNT* from $GPR[3]$ and $GPR[4]$ to $PRF[3]$ and $PRF[4]$ since $RNT[3] = 3$ and $RNT[4] = 4$. The target operand is allocated a new physical register; for example, we know that $PRF[5]$ is currently unused (i.e., no architectural register currently maps to it) so we allocate and use it, then update so that afterwards $RNT[2] = 5$. The result of this is that source operands in instruction #1 are mapped using *RNT* from $GPR[2]$ to $PRF[5]$. The target operand is again allocated a new physical register so that this time $RNT[5] = 6$. When instruction #2 has renaming applied we see that the "fake" output dependency, that would have existed between instructions #2 and #0 as a result of them both targeting the same register, is gone: the renaming process has allocated a new physical register for instruction #2 to use and hence the two can potentially execute at the same time.

Although use of a dedicated renaming table is conceptually the neatest way to see how register renaming works, in practise the ROB offers a simple way to implement the same functionality. Each instruction is already allocated a slot in the ROB when it is issued. Therefore, one can view the ROB slot as basically being the physical register that each instruction is allocated as a target.

9.3.3.3 Branch Unit (BU)

The example execution given for the scoreboarded design did not include any branch instructions; this masks the added complexity of branch management in a superscalar processor. As stated, the control strategy has to block all subsequent instructions from being issued when a branch is encountered. This ensures that there are no in-flight instructions which are invalid. In terms of implementation, one can think of this as an extra rule in the *ISU* stage which blocks issue whenever the *BU* is active.

However, in the context of the single-issue pipelined processor described earlier, we introduced the concepts of speculative execution and branch prediction as a means of improving performance versus simply leaving the processor idle while a given branch is resolved. On one hand, although matters are complicated by the increased number of in-flight instructions, the same techniques can be applied here. In part, this allows for much higher levels of performance because it permits the exploitation of ILP across branches. In more detail, the single-issue pipeline motivated us to use speculative execution in order that the pipeline was kept full of instructions. Performance in a superscalar processor is more or less dependent on keeping the PEs active: by using speculative execution we can actually overlap execution within PEs and exploit ILP in a way which was not possible with the single-issue pipeline. However, one could also point at potential drawbacks. Consider a single-issue pipelined processor X and a superscalar processor Y for example:

- In Y there are more computational resources (i.e., PEs) available than in X; as a result, any branch which stalls instruction fetch will waste relatively more resources on Y than on X. That is, since Y could do more work, having it sit idle is more costly in some sense.

- In Y there can potentially be more instructions in-flight at a given point in time than in X; as a result, an incorrect branch prediction within a speculative execution framework will need to do more work to squash invalid instructions and also waste relatively more resources.

Speculative execution can be easily supported by extending the ROB. The general idea is to add a flag to each ROB entry which determines if an in-flight instruction is being executed speculatively or not. Instructions with their speculative ROB flag set are not allowed to retire until the associated branch is resolved: if the speculation was correct, then they can complete as normal; if not, then they are simply squashed and removed from the ROB without writing any results.

9.3.3.4 Load/Store Unit (LSU)

Conceptually, we have so far viewed the Load/Store Unit (LSU) as simply another PE which performs memory accesses for us rather than computation. However, in reality the implementation of an LSU involves the solution of some subtle issues related to correctness and performance of said accesses. Internally, the LSU maintains two sets of buffers:

1. a set of load buffers into which pending loads are placed, and
2. a set of store buffers into which pending stores are placed.

which basically act in a similar way to reservation stations; in some descriptions, the LSU is treated as a special case PE without a dedicated reservation station.

A given load or store is performed as a two step process: first the effective address is calculated, then the actual memory access is performed. Any memory access is stalled until the effective address is available (perhaps computed by another PE) in the same way that an arithmetic instruction is stalled until all source operand values are available. However, a subtle difference is that the effective address is not communicated to the load or store buffers over the CDB: it has to be read from the register file. Since the ROB ensures in-order completion of instructions, this means effective addresses are completed in-order and hence that memory accesses are performed in-order: strong memory consistency is ensured.

Given that memory access takes a relatively long time, it is possible that the load or store buffers have several pending loads or stores in them a sub-set of which are ready to be performed. As such, there is an opportunity to re-order the accesses, i.e., revert to weak memory consistency, if it could be advantageous. For example, imagine a load X comes before another load Y in the program order but the effective address calculation for X takes much longer than Y for some reason: it could make sense to perform the loads out-of-order.

Crucially, if loads or stores use different effective addresses, then there is no dependency between them and no hazard will result from re-ordering them. One can think of this in the same terms as dependencies between instructions that access the register file, except that here we consider memory locations rather than registers.

If loads or stores use the same effective address, things are much more tricky. The most conservative approach is to perform accesses according to a non-speculative access scheme. The scheme waits until memory accesses are ready to be performed, i.e., the effective addresses are available, and then compares them with each other in a pair-wise manner. For two accesses X and Y where X is before Y in program order, either the effective addresses are different, in which case they can be re-ordered, or they are the same, in which case there are four possible situations:

X is a load, Y is a load In this case the two loads can be re-ordered as required: it does not matter in which order X and Y are performed, they still get the right value.

X is a store, Y is a load In this case, re-ordering X and Y would result in a RAW hazard since X would get the value that existed in memory before Y was executed rather than the value stored by Y. As such, Y has to be stalled until X is retired.

X is a load, Y is a store In this case, re-ordering X and Y would result in a WAR hazard since X would get the value stored by Y rather than the value that existed in memory before Y was executed. As such, Y has to be stalled until X is retired.

X is a store, Y is a store In this case, re-ordering X and Y would result in a WAW hazard since any load from the address after the two stores would get the value stored by X rather than that stored by Y. As such, Y has to be stalled until X is retired.

These rules can be relaxed by a speculative access scheme. Essentially this means that the LSU does not wait for the effective addresses of X and Y to both be available: it guesses that they are different. For example, where the effective address of X is not available but the effective address of Y is, it could make sense to speculatively re-order X and Y. It turns out that this only makes sense in the case where X is a store and Y is a load: if the speculation fails, i.e., X and Y turn out to use the same effective address, the processor can reissue Y to ensure it gets the right value. In other cases speculative re-order means reversing an actual memory access, an action which is seldom feasible.

9.3.3.5 Control Strategy

Using components detailed in Figure 9.21, execution of instructions is performed by the following control strategy:

Fetch (FET) and Decode (DEC) Instructions are fetched from memory. Note that since our aim is to the issue of more than one instruction per-cycle, the processor needs to fetch (and decode) more than one instruction per-cycle as well. In short, this means that the front-end must include a mechanism to keep up with the demand for instructions, otherwise the back-end will be starved of work.
Instructions are taken from the FET stage, decoded (so, for example, it is clear which source and target registers are used) into the issue buffer; the scoreboard tables are updated to include the instruction. Note that unlike the single-issue

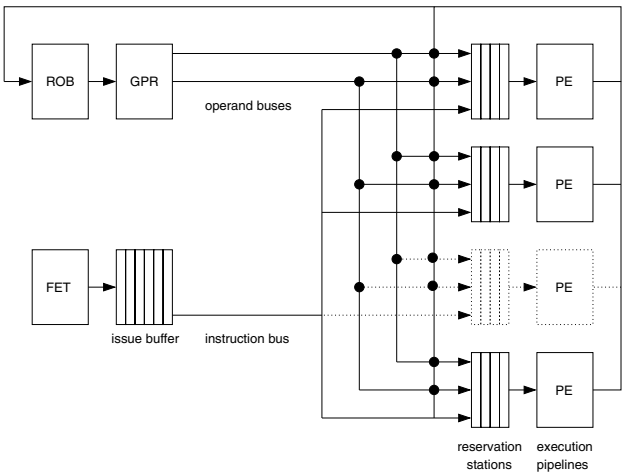


Figure 9.21 Step 2: a reservation station-based data-path with ROB.

pipeline described previously, the *DEC* stage does not fetch register operands from the register file; this task is deferred until later.

Issue (ISU) An instruction X in the issue buffer is stalled in the *ISU* stage until there is a reservation station slot free for some PE capable of processing X. When such a slot becomes free

- read any available operand values from the register file,
- allocate a slot at the tail of the ROB,
- issue the instruction into the free reservation station slot.

Dispatch (DIS) An instruction X housed in a reservation station slot stalls in the *DIS* stage until all operand values are available; the reservation station monitors the CDB for newly available results and fills them in as appropriate. When all the operand values for X are available, make a choice between potentially executable instructions within the reservation station and pass one of them into execution.

Execute (EXE) An instruction X then starts to be executed by the associated PE. As a result of passing through different PEs, different instructions might take different lengths of time to finish execution: a later instruction Y in program order might finish execution before X does.

Write (WRI) When execution of some instruction X is finished, write the result onto the CDB (so it can be used by any reservation station slots), and into the ROB entry. The slot occupied by X in a reservation station can now be made free.

Retire (RET) Once an instruction has finished execution, it stalls waiting until it reaches the head of the ROB buffer and is retired. The process of retirement depends on the instruction type:

- an arithmetic instruction will retire by writing results into the register file,
- a store instruction will retire by writing results into memory,
- a branch instruction will retire by either
  - squashing all other instructions in the ROB with their speculative execution flag set in the case where the branch was mispredicted (this implies the need to restart execution at the correct branch target), or
  - clearing the speculative execution flags of all other instructions in the ROB if the branch was correctly predicted.

Note that the control strategy has a slight inefficiency: sometimes an instruction will be issued to a reservation station in a state where it is immediately ready to be dispatched, i.e., with all operand values available. In this case the instruction pays a penalty of having to wait one cycle whereas in reality this could be improved upon by allowing a form of forwarding from the issue buffer, bypassing the reservation station, and straight into the associated PE. This basically allows such instructions to be issued and dispatched in the same cycle.

9.3.4 Further Improvements

9.3.4.1 Trace Caches

More usually used within the context of compiler design, the term **trace** is used to describe a non-cyclic sequence of instructions that *could* be executed consecutively. A given program will have many overlapping traces; a given trace may never be executed or executed very often based on the data-dependent nature of the program. To make this clearer, consider a short contrived program that takes two values held in $GPR[3]$ and $GPR[4]$ and sets $GPR[5]$ to the square of their maximum:

$$
\begin{array}{ll}
 & \ldots \\
 & GPR[2] \leftarrow GPR[3] - GPR[4] \\
 & \text{if } GPR[2] \leq 0, PC \leftarrow take \\
skip: & GPR[5] \leftarrow GPR[3] \\
 & PC \leftarrow done \\
take: & GPR[5] \leftarrow GPR[4] \\
done: & GPR[5] \leftarrow GPR[5] \cdot GPR[5] \\
 & \ldots
\end{array}
$$

In this case, control-flow can proceed through one of two traces: either it goes through instructions #0, #1, #2, #3 and #5 (i.e., the conditional branch is not taken, or $GPR[4]$ is the maximum) or through instructions #0, #1, #4 and #5 (i.e., the conditional branch is taken, or $GPR[3]$ is the maximum). Notice that the two traces overlap in that they both contain instructions #0, #1 and #5, and that the trace actually executed is data-dependent meaning we can only know at run-time which was executed.

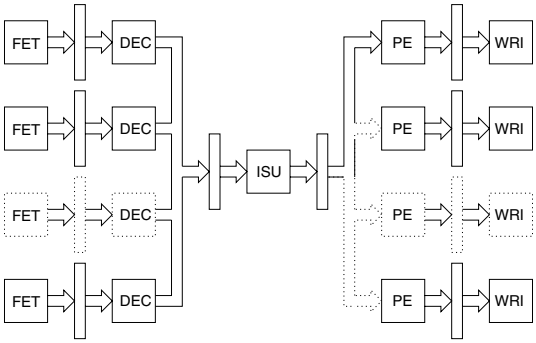


Figure 9.22 A conceptual diagram of a superscalar processor with hyperthreading.

The basic idea of a **trace cache** [58] is fairly simple. Such a device aims to capitalise on the fact that quite often, a small set of traces dominates execution. For example, with a program that contains a loop one would expect the trace representing instructions in the loop body to be executed many times. Of course, an instruction cache should capture such traces so that access to their constituent instructions is efficient. However, even if access to instructions in the cache is efficient, the processor still incurs the cost of passing them through potentially costly stages of fetch and decode. To reduce this cost the trace cache holds instructions from the working set of traces in a pre-decoded form, i.e., roughly in the form they would be in when decoding is finished. As a result, when an instruction fetch is required but the instruction is found in the trace cache, it can more or less bypass work normally performed by the processor front-end and progress straight into the issue buffer.

9.3.4.2 Hyperthreading

Even with a trace cache, there will be times when it is hard for the front-end to supply enough instructions to the back-end so that all execution pipelines are kept busy. For example, if a program performs a high proportion of memory accesses then the arithmetic pipelines may often be idle. This is problematic since it is the inherent nature of the program which causes under-utilisation of the execution pipelines, not really the processor.

As an approach to solving this problem, Intel introduced the concept of **hyperthreading** in their Pentium 4 range of processors. Hyperthreading is a specific example of a more general strategy for parallel processors termed **Simultaneous Multi-Threading (SMT)** whereby independent threads (i.e., program contexts) share the execution resources. The basic idea is to duplicate elements of the front-end that retain program state, essentially allowing several separate instruction streams to be fed into the back-end. By doing so, the hope is that the composed in-

struction stream dealt with by the back-end will be more diverse. That is, if stream #0 is composed mainly of memory access instructions and stream #1 mainly of arithmetic instructions, then their composition will avoid idle execution pipelines relative to only considering one or the other in isolation. Or if an instruction in stream #0 causes some stall condition, as the result of a cache miss for example, instructions from stream #1 can prevent the execution resources becoming idle. Figure 9.22 shows a conceptual diagram of a superscalar processor with hyperthreading; the diagram shows the general structure, in the specific example of the Pentium 4 there are just two duplicated parts of the front-end.

9.4 Vector Processors

9.4.1 Basic Concept

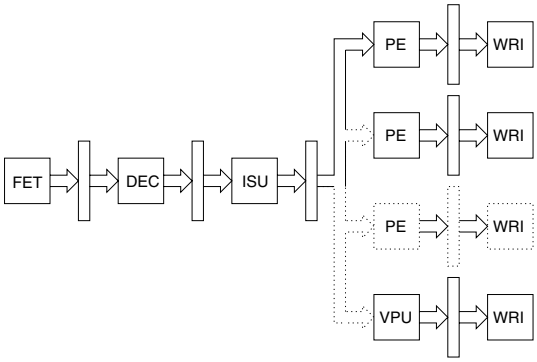
The processors we have examined so far operate on scalar data items and try to detect and capitalise on implicit ILP. An alternative approach is where ILP is made explicit somehow; the use of **vector processors** is one example of this approach. Conceptually, this is an easy step to make. In a scalar processor, the natural units of computation are single values held in registers. In a vector processor, one operates using registers that contain p separate values. Consider a simplistic example where we want to execute a vector addition:

$$A_i \leftarrow B_i + C_i, \text{ for } i \in \{0 \ldots p-1\}$$

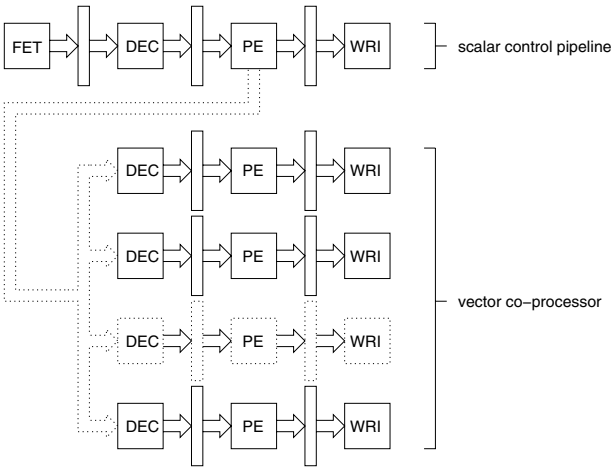
which we could implement in C using the function:

```
void add( int* A, int* B, int* C, int p )
{
  for( int i = 0; i < p; i++ ) {
    A[ i ] = B[ i ] + C[ i ];
  }
}
```

where `A`, `B` and `C` are all arrays of, say, 32-bit integers. Often, this sort of operation is termed a **pure vector** operation in the sense that computation is happening only component-wise: the i-th element of the result is generated using only the i-th elements of the inputs. Using Flynn's Taxonomy, this is clearly SIMD type parallelism. On a scalar processor we could implement the loop as follows:



(a) An integrated VPU approach.



(b) A VPU co-processor approach.

Figure 9.23 A conceptual diagram of two vector processors; the top design integrates a VPU as a standard superscalar PE, the bottom design opts for a dedicated VPU which is invoked by the execution stage of a scalar control processor.

$$
\begin{aligned}
& GPR[2] \leftarrow 0 \\
& GPR[3] \leftarrow p \\
loop: \ & \text{if } GPR[2] \geq GPR[3], PC \leftarrow exit \\
& GPR[4] \leftarrow MEM[\&\mathrm{B} + GPR[2]] \\
& GPR[5] \leftarrow MEM[\&\mathrm{C} + GPR[2]] \\
& GPR[6] \leftarrow GPR[4] + GPR[5] \\
& MEM[\&\mathrm{A} + GPR[2]] \leftarrow GPR[6] \\
& GPR[2] \leftarrow GPR[2] + 1 \\
& PC \leftarrow loop \\
exit: \ & \ldots
\end{aligned}
$$

But imagine that instead of our standard register file *GPR* which stores a single 32-bit value, we had a **vector register file** where each register stores p such values. Let $VR[i]$ denote the i-th **vector register** and $VR[i][j]$ denote the j-th value or **subword** in the i-th register. Clearly we also need some vector instructions capable of operating on values held in the vector register file. Equipped as such, the whole loop collapses into a few instructions:

$$
\begin{aligned}
&VR[4] \leftarrow MEM[\&\mathtt{B}+0]\ldots MEM[\&\mathtt{B}+p-1]\\
&VR[5] \leftarrow MEM[\&\mathtt{C}+0]\ldots MEM[\&\mathtt{C}+p-1]\\
&VR[6] \leftarrow VR[4]+VR[5]\\
&MEM[\&\mathtt{A}+0]\ldots MEM[\&\mathtt{A}+p-1] \leftarrow VR[6]\\
&\ldots
\end{aligned}
$$

Of course, each vector instruction is hugely complex compared to the equivalent scalar instruction; in some sense, one might view a vector processor as an embodiment of CISC design. Important examples of this complexity are the vector add instruction which uses $VR[i]$ and $VR[j]$ as source operands and produces a result in $VR[k]$ essentially computing p additions, and the vector load and store instructions which load or store p values from or to memory starting at some base address.

One might classify the technique as using **static scheduling**, since the specification of parallelism is done at compile-time, and **single-issue**, since the processor issues one instruction per-cycle (even though each instruction implies many similar operations). There are several important benefits that result:

1. Normal work over vector operands (i.e., the loop) contains branches; without them, vector programs have fewer control dependencies.
2. Since parallelism within vector operations is explicit, less hardware is required to manage issues of data dependency and effectively exploit the parallelism.
3. The vector program captures the entire loop in a few instructions; the burden on fetch and decode stages is therefore reduced.
4. Memory access patterns resulting from vector loads and stores are much more regular (e.g., vector elements are contiguous in memory).

This topic has a rich history [20]; a (very) rough time line of interesting advances starts around 1960 when development of the ILLIAC IV computer started at the University of Illinois, based on previous research by Westinghouse Electric. When delivered the computer had 64 parallel ALUs and could reach up to 100 MFLOPS.

9.4.2 A Dedicated Vector Processor

There are numerous design options for vector processors. For example, a register-register design is similar to a load-store RISC design: one cannot mix operands from registers and memory. In contrast, a memory-memory design uses operands that are always loaded from and stored into memory. Consider a naive design for a vector processor based on a register-register approach: from an initial scalar processor

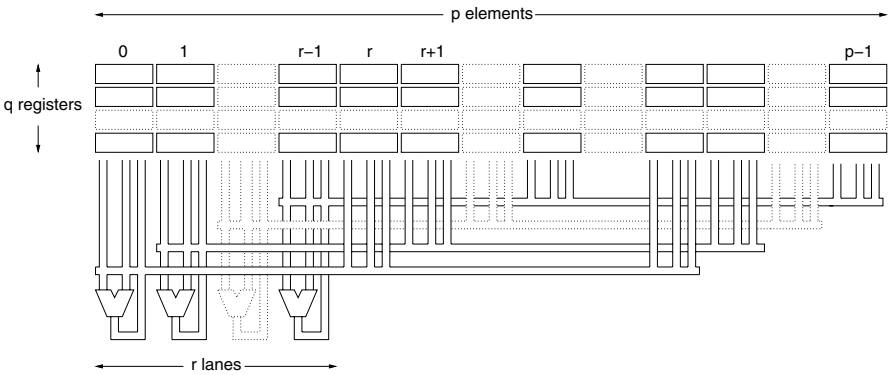


Figure 9.24 A strided (i.e., $r < p$) vector register file.

we would replace $GPR[i]$ with $VR[i]$ that holds p values and replicate the ALU p times to cope with p-element operands. There are some problems with this this. For example, what value do we select for p, some programs might want $p = 8$ while others might prefer $p = 128$. If p is even somewhat large, the amount of logic required for the replicated ALUs will be problematic ! Further, not all operations we want to perform will be vector operations; for example, there might necessarily be some scalar outer loop over a number of vector instructions, the loop is controlled by scalar instructions. One design option which addresses these problems does so by separating the architecture in two parts: a **Vector Processing Unit (VPU)**, which is somewhat analogous to a separate **Floating Point Unit (FPU)** used to support early processors, and a conventional scalar processor which controls the VPU and interfaces with main memory.

We already know how to design the scalar processor, obviously the VPU is the most interesting part; if the scalar processor is a conventional pipeline, when the execute stage deals with scalar instructions as normal but hands-off vector instructions to the VPU. As shown in Figure 9.24, a vector register file holds values ready to be used in execution; say each element in a vector is n bits, there are p such elements in each vector and there are q such registers. Roughly speaking the vector register file is a $(p \times q)$-element array of n-bit elements. The VPU houses a number, say r, of PEs which process values from the vector registers. Sometimes these parallel PEs are called **lanes** and ideally we have $p = r$ so that PE i processes element i of the source operands and produces element i of the result. Additional complexity is introduced when $r < p$ (we limit this to cases where $r|p$). In this case PE i has to process elements i, $i + r$, $i + 2r$, ... and r is termed the **stride**. Roughly speaking, the VPU has to iterate a single vector operation $\lceil p/r \rceil$ times to generate all p elements of the result. This means that each instruction needs multiple cycles to finish execution and also that the register file must support access to the sub-words for a given iteration.

The VPU itself can be pipelined so operand fetches from the vector register file, execution of vector operations and writes to the vector register file can be overlapped. However, we need a way to interface the vector register file to memory somehow in order to allow vector load and store operations. In viewing the VPU as entirely separate, this is problematic. More likely, the VPU shares the load/store unit in the scalar processor which already has a memory interface. The load/store unit clearly needs to iterate p atomic load or store operations to cope with vector content; the concept of a burst mode in the memory interface can accelerate load/store performance. Each lane might also be equipped with a small cache to accelerate memory accesses, i.e., avoid using the load/store unit; the regular nature of accesses to vectors held in memory means even simple cache designs can be highly effective. We might also want to allow communication between lanes as well as to and from memory so as to support non-pure vector operations more efficiently.

9.4.3 SIMD Within A Register (SWAR)

Constructing a dedicated vector processor is quite an investment; one needs quite a bit of extra hardware for computation and also control. Modern scalar processors also exploit vector processing techniques in a backwards compatible manner. The idea is to implement ISA and micro-architectural changes that allow one to view a normal register as a number of sub-words: think of it as adding the VPU alongside an existing ALU, or as a vector execution pipeline in a superscalar design.

This is typically realised by packing sub-words together into a large value and utilising appropriate arithmetic on them; the more sub-words we process in parallel the bigger the improvement in performance. To fix notation, if x is a packed value comprising p sub-words we use x_i to denote sub-word i. The general idea is to compute all the results

$$r_i = x_i \odot y_i$$

such that given an instruction $\odot$, the ALU computes all p results, potentially in parallel if there are enough computational resources.

Media applications have driven development of SWAR within desktop class processors and as a result we now have several generations of MMX and SSE by Intel; VIS by Sun; 3DNow! by AMD; and AltiVec by Apple, IBM and Motorola. MIPS have a basic SIMD extension called MIPS-3D which is intended for use on floating point values in 64-bit processors; to aid continuity we discuss MIPS-3D here briefly even though it is slightly out of the scope of the reduced, integer-only MIPS32 ISA. MIPS-3D revolves around a data-type called paired-single (PS) that packs two 32-bit floating point values into a single 64-bit register. The idea is that MIPS-3D offers two-way SIMD, i.e., it sets $p = 2$, and allows operation on a pair of values using a single instruction.

MIPS-3D is not a good example of a SIMD instruction set however; it is very limited in terms of the actual operations it can perform. Deviating again from our

```
void add( int* A,
          int* B,
          int* C, int p )
{
  int a, b, c;

  for( int i = 0; i < p; i = i + 1 )
  {
    b = B[ i ];
    c = C[ i ];

    a = b + c;

    A[ i ] = a;
  }
}
```

Listing 9.3 A sequential vector addition function.

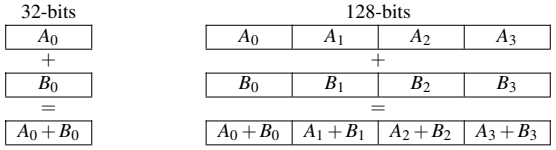
```
void add( int* A,
          int* B,
          int* C, int p )
{
  __m128i a, b, c;

  for( int i = 0; i < p; i = i + 4 ) {
    b = _mm_load_si128( B + i );
    c = _mm_load_si128( C + i );

    a = _mm_add_epi32( b, c );

    _mm_store_si128( A + i, a );
  }
}
```

Listing 9.4 A SIMD parallel vector addition function.

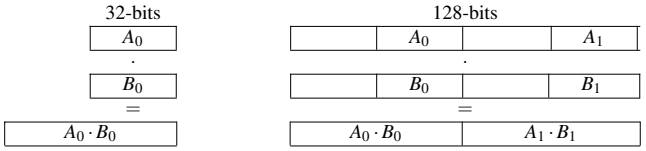
use of MIPS32 as an example, Listing 9.3 and Listing 9.4 are specific to the Intel Pentium range of processors; they offer a better concrete example. As before, we are adding two p-element vectors B and C together to form A as a result. This is a component-wise operation, i.e.,

$$A_i = B_i + C_i, i \in \{0 \ldots p-1\}.$$

As such, we can write a basic loop, as in Listing 9.3, to iterate over the elements of B and C and produce the corresponding element of A. Ignoring the overhead of the loop, this takes $O(p)$ instructions, one addition for each value of i. Listing 9.4 parallelises this basic loop: since the computation for all values of i are independent from each other, we can do them at the same time. It performs four at once by using the SIMD instruction set in the processor; the code uses the intrinsics feature of GCC to access the instruction set through what look like function calls. First four elements of each of B and C are loaded into `b` and `c` which are given a special data-type. Then, a single addition instruction is executed which performs a SIMD



(a) A pure vector addition.



(b) A non-pure vector multiplication.

Figure 9.25 Two figures showing the difference between pure and non-pure vector arithmetic.

addition to the elements in `b` and `c`; finally the result is stored back into A. Again ignoring the loop overhead, this version only takes $O(p/4)$ instructions; we expect it to execute roughly four times faster as a result.

9.4.4 Issues of Vectorisation

The act of turning a scalar program into a vector equivalent is often termed **vectorisation**. This is made more difficult by two features of most SWAR enabled processors. Firstly, and most obviously, not all operations are pure vector operations: the processor must support operations like swizzles (a permutation of sub-words) and packs (an interleaving or zip of sub-words). Secondly, some operations require non-standard semantics. For example, an $(n \times n)$-bit multiplication produces a $2n$-bit result. Storage of this result breaks the normal component-wise nature of pure vector operations as described in Figure 9.25: essentially the data-path width fans out from n-bit to $2n$-bit in order to accommodate the result.

Designs use different approaches to solving such issues, but generally speaking non-pure vector operations mean more data reorganisation which is overhead versus useful computation. We remark below on two concrete examples of where vectorisation is difficult, and potential solutions that resolve this difficulty.

9.4.4.1 Masked Versus Explicit Control-Flow

Managing control-flow in vector programs can be a challenge. On a scalar processor we simply use a branch instruction to move the *PC* to the correct part of the program. However, on a vector processor, one can easily imagine a requirement to perform one computation on one element of the vector and another computation on another element. Consider a concrete example; given an 8-bit byte value a, a particular stage of the AES block cipher involves computing the function:

```
uint8 xtime( uint8 a )
{
  if( a & 0x80 )
    return 0x1B ^ ( a << 1 );
  else
    return        ( a << 1 );
}
```

If you consider a as a member of $\mathbb{F}_{2^8}[x]$, then the left-shift is performing multiplication by x and the conditional XOR performs reduction modulo $x^8+x^4+x^3+x+1$. AES works on a vector of sixteen such values; the problem is, how do we vectorise the `xtime` function to operate on an entire vector at once ? For example, it is clear that one component of the vector might require a different control-flow path through `xtime` than another: this is not a pure vector operation.

Most SWAR designs attack this problem by providing pure vector comparison with slightly non-standard semantics. Usually, we think of logical false as 0 and true as 1 or, in C, false as 0 and true as any non-zero. In a SWAR design with, say, 8-bit sub-words we are free to view false as $00_{(16)}$ and true as $FF_{(16)}$. This allows us to use a masking approach to implement the body of `xtime`. Imagine $VR[1]$ holds the input vector; we compute the result into $VR[2]$ using the program fragment

$$\begin{aligned}
VR[2] &\leftarrow VR[1] \wedge 80_{(16)} \\
VR[1] &\leftarrow VR[1] \ll 1 \\
VR[2] &\leftarrow VR[2] = 80_{(16)} \\
VR[2] &\leftarrow VR[2] \wedge 1B_{(16)} \\
VR[2] &\leftarrow VR[1] \oplus VR[2]
\end{aligned}$$

Imagine $VR[1][0]$ contains a value which takes the first control-flow path through `xtime`, i.e., includes the conditional XOR, whereas $VR[1][1]$ takes the bottom control-flow path. Instruction #0 performs a component-wise AND with the constant $80_{(16)}$; for $VR[1][0]$ this will result in $80_{(16)}$ but for $VR[1][1]$ the result is $00_{(16)}$. Instruction #1 performs a component-wise left-shift; for both $VR[1][0]$ and $VR[1][1]$ the value is shifted left. Instruction #2 performs a component-wise comparison with the constant $80_{(16)}$; because of the way the SWAR comparisons work, this means for $VR[2][0]$ the result is $FF_{(16)}$ but for $VR[2][1]$ the result is $00_{(16)}$. Next instruction #3 performs a component-wise AND with the constant $1B_{(16)}$; for $VR[2][0]$ the result is $1B_{(16)}$ but for $VR[2][1]$ the result is $00_{(16)}$. Finally, we take the shifted result in $VR[1]$ and perform a component-wise XOR with $VR[2]$. The crucial thing to note is that as a result, the shifted value in $VR[1][0]$ is XORed with the constant $1B_{(16)}$ as

required by the first control-flow path whereas $VR[1][1]$ is XORed with $00_{(16)}$, i.e., is unaltered as required by the second control-flow path. This has all been achieved without per-sub-word conditional operations.

9.4.4.2 Normal Versus Saturated Arithmetic

A similar problem to control-flow occurs when considering any exceptional condition such as carry or arithmetic overflow. On a scalar processor, if some operation overflows, the processor detects the condition and then acts accordingly; it might set a flag in the status register. However, on a vector processor, we are performing p operations at once: what happens if one computation overflows and another does not; how do we detect the condition and how do we act accordingly ?

Typically, the answer is not to replicate the status register, i.e., one does not have p overflow flags which are individually testable. Instead, alternate forms of arithmetic are used which cope gracefully with exceptional conditions; examples in the context of overflow are so-called **clamped addition** or **saturating addition**. For an n-bit saturating addition denoted by $\uplus$ we have that

$$x \uplus y = \begin{cases} x+y & \text{if } x+y < 2^n \\ 2^n - 1 & \text{otherwise} \end{cases}$$

so instead of an overflow wrapping around, the result is locked to the maximum possible value. This seems like an odd choice of "graceful" but it is often useful in the context of media applications. Consider a concrete example; given a vector of 8-bit bytes which represent an image as pixel values, we could brighten the pixels using the function:

```
void brighten( uint8* A, uint8* B, int p )
{
  for( int i = 0; i < p; i++ )
    A[ i ] = B[ i ] + 1;
}
```

However, if a pixel is already as bright as it can be and we use a modular type addition, the pixel goes from bright white to dark black ! Using a clamped addition, the pixel stays bright white; it does not get brighter but the human eye probably does not notice. Again, this has all been achieved without per-sub-word conditional operations.

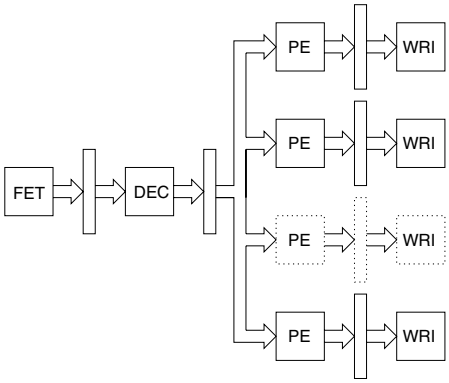


Figure 9.26 A conceptual diagram of a VLIW processor; the non-dashed path between *ISU* and the PEs is supposed to show that a given instruction bundle is split into parts with each part passed to one PE.

9.5 VLIW Processors

9.5.1 Basic Concept

By definition, for pure vector operations, a vector processor applies the same operation to each element of the source operands. However, this is sometimes a bit too much of a constraint: often, programs do not map well onto pure vector operations and even if they do, one might not get the best resource utilisation. Superscalar processors do not impose this constraint; one feeds in a stream of arbitrary instructions and the hardware works out which can be executed in parallel. The scheme usually gives good resource utilisation on well-scheduled code but if we already know which instructions can be executed in parallel, all that hardware is a bit of a waste.

Very Long Instruction Word (VLIW) processors [21] can be thought of as sort of a compromise between the two. That is, in a superscalar processor, one instruction specifies one operation whereas in a VLIW processor, r operations are merged into one instruction **bundle**; operation i is sometimes termed the i-th **channel** or **lane**. If one instruction is 32 bits long, the instruction bundle could be up to $32r$ bits long (although fancy encoding schemes may reduce this). The crucial thing to realise is that the bundle format is fixed: one cannot create a bundle of arbitrary instructions, the hardware resources dictate the bundle composition. For example, we might say it has to specify two integer operations, one multiply operation and one memory access operation.

One might classify this technique as using **static scheduling**, since the specification of parallelism is done at compile-time, and **multiple-issue**, since the processor

issues many instructions per-cycle (even though they may be packed into a single bundle).

9.6 Further Reading

- S.P. Dandamudi.
  Guide to RISC Processors.
  Springer-Verlag, 2005. ISBN: 0-387-21017-2.
- A. Grama, A. Gupta, G. Karypis and V. Kumar.
  Introduction to Parallel Computing.
  Addison-Wesley, 2001. ISBN: 0-201-64865-2.
- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.
- J.L. Hennessy and D.A. Patterson.
  Computer Architecture: A Quantitative Approach.
  Morgan-Kaufmann, 2002. ISBN: 1-558-60724-2.
- M.D. Hill, N.P. Jouppi and G.S. Sohi.
  Readings in Computer Architecture.
  Morgan-Kaufmann, 2000. ISBN: 1-558-60539-8.
- J. Stokes.
  Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture.
  No Starch Press, 2006. ISBN: 1-593-27104-2.
- A.S. Tanenbaum.
  Structured Computer Organisation.
  Prentice-Hall, 2005. ISBN: 0-131-48521-0.

9.7 Example Questions

37. The figure below shows a block of combinatorial logic built from six parts where the name and latency of a given part is displayed inside it; the last part is a register which stores the result:

Part A 40ns	Part B 10ns	Part C 30ns	Part D 10ns	Part E 50ns	Part F 10ns	REG 10ns

It is proposed to pipeline the block of logic using two stages such that there is a pipeline register in between parts D and E:

Part A 40ns	Part B 10ns	Part C 30ns	Part D 10ns	REG 10ns	Part E 50ns	Part F 10ns	REG 10ns

a. Explain the terms latency and throughput in relation to the idea of pipelining.

b. Calculate the overall latency and throughput of the initial circuit described above.
c. Calculate the overall latency and throughput of the circuit after the proposed change.
d. Calculate the number of extra pipeline registers required to maximise the circuit throughput; state this new throughput and the associated latency. Explain the advantages and disadvantages of this change.

38. a. It has been suggested that in designing a new processor, a pipelined approach should be used instead of a multi-cycle approach. Explain how and why a pipelined design would improve performance; draw a data-path block diagram to describe which components fit into which pipeline stages and why.
b. In the context of pipelined processor design, explain the meaning of the following terms and give examples of each:

   i. Structural dependency.
   ii. Control dependency.
   iii. Data dependency.

c. A pipelined processor might stall when such dependencies are encountered; the performance is decreased as a result. Explain how the three types of dependency listed above can be avoided so that stalls are minimised.
d. In a pipelined processor design, branches can cause control dependencies that stall the processor until resolved. There are several options for dealing with this problem; select **two** of the **three** options below and describe how they work:

   i. Branch delay slot.
   ii. Predicated execution.
   iii. Branch prediction.

   For each option, your answer should explain how the scheme works, how it reduces the performance impact, and the trade-off between complexity and performance for the programmer and/or in the hardware.

39. Consider the following C fragment which adds together two n-element vectors B and C to produce a result A where each element is a 32-bit integer:

```
for( int i = 0; i < n; i++ ) {
  A[ i ] = B[ i ] + C[ i ];
}
```

The fragment can be parallelised since each iteration of the loop is independent. Explain how this could be exploited using

a. A vector processor.
b. A VLIW processor.

40. a. Using a diagram and an example program if appropriate, describe the motivation for and design of (i.e components in) a vector processor.

b. Describe **two** reasons why use of a vector processor may improve performance (for suitably written programs) versus a scalar processor.
c. The algorithm below describes the core loop of RC6, a block cipher. The algorithm operates in-place on an input of four values $A \ldots D$ and an array K. The values $A \ldots D$ and each $K[i]$ are 32-bit unsigned integers; the loop includes an operation $rotl(x,y)$ which performs bitwise left-rotate of x by a distance y (where the distance is constrained to be $0 \ldots 31$, i.e., we just use $y_{4\ldots0}$).

```
1  RC6-LOOP(A,B,C,D,K)
2  begin
3      for i = 0 upto 44 step 2 do
4          t0 ← rotl(B · ((2 · B) + 1), 5)
5          t1 ← rotl(D · ((2 · D) + 1), 5)
6          A ← rotl(A ⊕ t0, t1) + K[i + 0]
7          C ← rotl(C ⊕ t1, t0) + K[i + 1]
8          t2 ← A
9          A ← B
10         B ← C
11         C ← D
12         D ← t2
13 end
```

i. The Intel Pentium 4 processor allows a form of vector processing called SIMD Within A Register (SWAR); it packs four 32-bit sub-words into each 128-bit register. Stating any reasonable assumptions (for example about the instruction set) and using either a diagram or pseudo-code, write a vectorised implementation of the straight-line RC6 loop body.
ii. Stating any reasonable assumptions, estimate the performance improvement as a result of vectorisation. Comment on any reason(s) your result is lower than the theoretical limit, and any changes you might make to the processor design or instruction set to solve this problem.

41. a. In a superscalar processor, performance is reliant on the front-end supplying enough instructions to keep computational resources in the back-end busy. Explain how and when speculative execution helps to solve this problem.
b. Speculative execution is often based on prediction of branch behaviour (i.e., if a branch is taken and what the target address is). One can consider at least two cases:

i. static branch prediction,
ii. dynamic branch prediction.

For **each** case, describe a mechanism to predict branch behaviour; take care to describe the structures within the processor, how they are used and any advantages or disadvantages of the mechanism.
c. Imagine that on some processor function calls are always realised using dedicated "call" and "return" instructions instead of generic branches.

i. Using an example and stating any reasonable assumptions, show how such instructions would be used to implement a function call.
ii. Outline a mechanism that could provide more accurate branch target prediction for these instructions than in those from the previous question.

Part III
The Hardware/Software Interface

Chapter 10
Linkers and Assemblers

Programs must be written for people to read, and only incidentally for machines to execute.

– H. Abelson and G. Sussman

Abstract We have already seen that processors execute programs that are stored in memory as binary values according to a prescribed encoding. Viewed in this way we usually say that the program is written in **machine code** in the sense that it is suited for interpretation by the machine: it is hard (or at least tedious) for a human to read or write even trivial programs in this form. In this chapter we investigate the use of low-level software tools to assist the programmer by automating various aspects of software development. We focus on two such software tools: **linkers**, which translate object files into executable files, and **assemblers**, which translate programs written in low-level languages into object files.

10.1 Introduction

The basic purpose of a **compilation tool** is to translate a program written in some language, typically containing high-level constructs, into a low-level or executable form. Essentially this mechanises the processes that an expert programmer might perform by hand. To allow successful translation, the programmer specifies the program using a precise list of operations and symbols on which those operations act. In modern terms it is unusual for a compiler or assembler to exist in isolation from other tools. More commonly, they form part of a **compiler tool-chain** which performs the translation process step by step. Roughly speaking, at each step the tool-chain translates a source program described at a higher-level (by which we mean a description that is natural and expressive) into a lower-level target program (by which we mean a description more bound by constraints of the target platform). Although there is some overlap in definition, we can identify three main tools within the standard tool-chain detailed in Figure 10.1: **compilers**, which translate programs

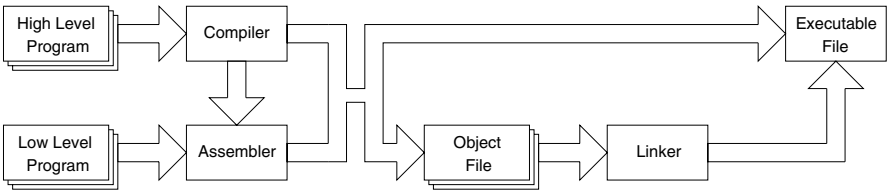


Figure 10.1 A rough block diagram of tool-chain components.

written in high-level languages into programs written in low-level languages or directly into object or executable files; **assemblers**, which translate programs written in low-level languages into object or executable files; and **linkers**, which combine object files into executable files or other object files. Omitted from this list is the **loader** tool which, as part of the operating system, is tasked with taking an executable file and loading it into memory ready for execution. Although there are overlaps with what the linker does, we defer explanation of the loader until Chapter 12.

Wherever used, compiler tools can offer several benefits. Specifically, by allowing expression of programs more naturally and by offering automated forms of analysis, optimisation and transformation, a programmer will typically improve their productivity, reduce their rate of error and generally produce software of a higher quality. Additionally, the combination of several tools rather than one monolithic tool is attractive for several reasons, mainly related to modularity and reuse of software. In particular, it allows us to split a program into parts and process the parts in different ways using different components of the tool-chain:

- If the program is split into parts, collaborative development is made much easier: each programmer can work on a separate part of the program and the parts can be joined together during the compilation process.
- If a particular part is changed (to add functionality or fix a problem for example) only that part need be reprocessed by the higher-level parts of the tool-chain. For example, perhaps only that part need be recompiled (although any resulting executable would also need to be regenerated to include the change).
- If we need to write one part in a particular programming language (e.g., in C) and another part in a different language (e.g., assembly language), we can use different tool-chain components to produce object files that can then be linked together.
- If we need to reuse a particular part in another program, we can extract that part more easily. For example we might construct a collection of commonly used functions into a **library** which the linker combines with other programs.

A definitive history of programming languages and compiler tools is beyond the scope of this book, and is in fact difficult to capture at all due to the breadth of

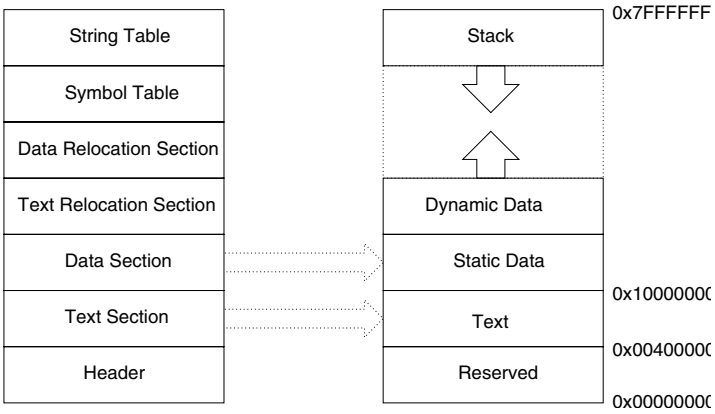


Figure 10.2 The seven `a.out` sections (left), and the MIPS32 memory model (right) onto which they map.

the subject. To cope, we dedicate this chapter to the introduction of linkers and assemblers and defer discussion of compilers until Chapter 11.

It is often hard to separate discussion of a programming language from an associated tool: sometimes they might refer to the same thing but more often there might be many compiler implementations for a given language. Even so, it is easy to see that assembler-like tools existed in the 1940s and early 1950s for computers such as the ENIAC and EDSAC; essentially these acted as a means of writing machine code in a human-readable form. Invention of the first higher-level compiler tools is usually attributed to Grace Hopper in 1952. While working for the US Naval Reserve on the UNIVAC computer, Hopper developed a program called **A-0** which translated symbolic mathematical expressions and functions into machine code. Around the same time, various incarnations of a language and compiler called **Autocode** were developed by researchers working on the SSEM [7, 8] computer. Both were quickly extended to include higher-level concepts. A-0 was followed by B-0 (or FLOWMATIC) which translated more expressive programs into machine code and which eventually spawned the COBOL programming language. A version of Autocode was developed to improve on the ALGOL programming language and support the Atlas computer. By this time the language supported truly high-level feature including various control-flow and conditional statements, and compilation of functions and complex mathematical expressions.

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  int dot( int*, int*, int );
5
6  int* A;
7  int* B;
8
9  int main( int argc, char* argv[] )
10 {
11   int n = atoi( argv[ 1 ] );
12
13   A = ( int* )( malloc( n * sizeof( int ) ) );
14   B = ( int* )( malloc( n * sizeof( int ) ) );
15
16   for( int i = 0; i < n; i++ ) {
17     A[ i ] = atoi( argv[ i+  2 ] );
18   }
19   for( int i = 0; i < n; i++ ) {
20     B[ i ] = atoi( argv[ i+n+2 ] );
21   }
22
23   int a = dot( A, B, n );
24
25   printf( "the dot product is %d\n", a );
26 }
```

Listing 10.1 The source file P_{SRC}, part of a C program to compute a vector dot product that demonstrates usage of different sections of the memory model.

10.2 The Memory Model

A program is a static description: we execute a program within an environment or context, and term a program in execution a **process**. One of the features of the environment in which a process executes is the layout of the address space (i.e., the valid memory addresses) it can access. So far we have dealt only with execution of one process at a time; this means that the process has unconstrained access to the entire address space, a simplification we will continue with until Chapter 12. However, it is important to introduce the concept of a **memory model** which is imposed by both the operating system and underlying architecture: the process must work within the rules of the memory model by placing various program components at well-defined areas in the address space.

The right-hand side of Figure 10.2 details the MIPS32 memory model as four main sections. The operating system reserves the space from address $00000000_{(16)}$ through to $003FFFFF_{(16)}$; a program is loaded into the **text section** starting at address $00400000_{(16)}$. As we described earlier in Chapter 4, this presents a problem if the program is written with the assumption it will be loaded at address $00000000_{(16)}$. In this case it requires **relocation**, a topic we examine later. Following the text section is the **data section** which holds static and dynamic data objects used by the program; the dynamic data grows upwards from low addresses toward high addresses. Finally, at the other end of the address space, the **stack section** starts at address $7FFFFFFF_{(16)}$ and grows downward.

```
int dot( int* X, int* Y, int n )
{
  int a = 0;

  for( int i = 0; i < n; i++ ) {
    a += X[ i ] * Y[ i ];
  }

  return a;
}
```

Listing 10.2 The source file Q_{SRC}, part of a C program to compute a vector dot product that demonstrates usage of different sections of the memory model.

While the text section is populated with instructions from the executable file, the various data sections are used in different ways to implement different features in a program. To describe these features, we use a single example program split into two source files which we refer to as P_{SRC} and Q_{SRC} and detail in Listing 10.1 and Listing 10.2. In combination, the two source files compute the dot product of two n-element vectors A and B

$$a = \sum_{i=0}^{n-1} A_i \cdot B_i$$

where the C variables `A`, `B` and `n` are read from the command line so as to represent A, B and n, and then the result `a` is displayed for the user.

10.2.1 Stack Section

Traditionally, a **stack** is a **First-In Last-Out (FILO)** structure. We can model the behaviour of a stack in memory using two functions

$$\begin{aligned} \text{PUSH}(x,y) &: \{0,1\}^* \times \{0,1\}^n \rightarrow \{0,1\}^* \\ \text{POP}(x) &: \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^n \end{aligned}$$

whose behaviour is defined by

$$\begin{aligned} \text{PUSH}(x,y) &= x \text{ s.t. } x \leftarrow x-1, MEM[x] \leftarrow y \\ \text{POP}(x) &= (x,y) \text{ s.t. } y \leftarrow MEM[x], x \leftarrow x+1. \end{aligned}$$

In this context x is the **Stack Pointer** (*SP*), an address maintained in order to keep track of the **Top of Stack** (*ToS*). There are actually two ways to talk about *ToS*: either it is the *next free* entry on the stack or the *first full* entry. Some ISAs define *SP* as a special-purpose register with associated instructions to access it; in other processors *SP* is part of the general-purpose register file and can be accessed like any other. MIPS32 takes the latter approach, reserving $GRP[29]$ for use as *SP*. Some ISAs, such as ARM, include dedicated instructions for dealing with the two choices

and also whether the stack grows upward or downward. Here we assume *ToS* always points to the first full entry and that it grows downward in memory starting at some known address.

The PUSH and POP functions act in a similar way to STORE and LOAD except as well as updating or retrieving an element from memory, they also update the value of the stack pointer: PUSH puts y onto the top of the stack and returns the updated stack pointer, POP takes the value from the top of the stack and returns it with the updated stack pointer. Note that typically there are some conditions on PUSH and POP to prevent erroneous use, for example pushing items onto the stack if it is full or popping them off if it is empty; we do not address this problem here. Further, note that we have only considered pushing and popping elements that match the memory resolution; clearly one can push and pop larger elements with repeated calls to PUSH and POP.

Of course, how much of this detail we actually need to worry about depends on the level at which we are programming. Although the stack is a dynamic data structure which grows and shrinks under control of the program, we do not really see this at a high-level. For example if we write a C program, the C compiler emits instructions that control the stack; we do not interact with it directly. On the other hand, at a low-level we can control the stack directly by manipulating *SP*: if we write an assembly language program, we take on full control of the stack and need to worry about how to realise PUSH and POP.

Consider a small example. Say we initially set $x = 3$ and assume the memory is zeroed, i.e., that $MEM[0\ldots2] = (0,0,0)$ for example. Calling PUSH and POP with some values gives a behaviour such as

$$\begin{array}{ll}
x = \textsc{Push}(x,4) \rightarrow x = 2, & MEM[0\ldots2] = (0,0,4) \\
x,y = \textsc{Pop}(x) \rightarrow x = 3,\ y = 4, & MEM[0\ldots2] = (0,0,4) \\
x = \textsc{Push}(x,3) \rightarrow x = 2, & MEM[0\ldots2] = (0,0,3) \\
x = \textsc{Push}(x,2) \rightarrow x = 1, & MEM[0\ldots2] = (0,2,3) \\
x,y = \textsc{Pop}(x) \rightarrow x = 2,\ y = 2, & MEM[0\ldots2] = (0,2,3) \\
x,y = \textsc{Pop}(x) \rightarrow x = 3,\ y = 3, & MEM[0\ldots2] = (0,2,3) \\
x = \textsc{Push}(x,1) \rightarrow x = 2, & MEM[0\ldots2] = (0,2,1) \\
x,y = \textsc{Pop}(x) \rightarrow x = 3,\ y = 1, & MEM[0\ldots2] = (0,2,1)
\end{array}$$

Notice that the memory content is not cleared when we call POP: although the call returns the right value, this is also left in memory until overwritten.

10.2.2 Static Data Section

The static data section is actually comprised of two parts: initialised and uninitialised data. Both are, in some sense, used to store global data items within the program. Within P_{SRC} the string constant

```
the dot product is %d\n
```

is a good example of data that would be allocated space in the initialised data section. When the executable file is loaded into memory, the string is allocated space and the actual content is copied into that space in order to initialise it ready for use during execution. However, the pointers `A` and `B` are not placed in this section: some space clearly needs to be allocated to these global data items, but they need not be populated using information in the executable file. As a result, they would be allocated space in the uninitialised data section.

10.2.3 Dynamic Data Section

The dynamic data section, sometimes termed the **free store** but which we will refer to using the more colloquial term **heap**, is perhaps the most difficult to describe. In part this is because the concept is so abstract: unlike the stack, the processor does not typically mandate any rules or provide any support for managing the heap. As a result, management of dynamic data is highly dependent on the programming language and operating system used. The basic idea is that sometimes we do not know at compile-time how much space will be needed for some aspects of a program. The program represented by P_{SRC} and Q_{SRC} contains a good example of this: the size of arrays `A` and `B` are not known until the program is executed and the user decides on a value of `n`. As such we need to delay allocation of space for `A` and `B` until run-time, an act which is achieved by using the heap.

In the context of a C program, the heap is typically managed by an implementation of the `malloc` library (and variants thereof). These are library functions like any other, such as `printf` for example, and form part of the program rather than part of the operating system. The basic idea is that the C program requests a new block of memory by calling `malloc`, potentially calling `free` later when it is no longer required. The library is tasked with managing the heap so that these requests are correctly satisfied; essentially this means keeping track of the allocated blocks within the area of memory assigned to the heap, and performing their allocation in a sensible way. This task is crucial and presents a number of challenging problems:

- Finding an effective algorithm for management of the heap is a challenge because it usually is not possible to predict how it will be used at compile-time: we cannot tell what sequence of calls to `malloc` and `free` will be used, let alone the size of blocks which need to be allocated. This means the algorithms need to be generic and use local heuristics to ensure their global operation is effective.
  Put more simply, such algorithms will need to make decisions at a given point in time which may or may not turn out to be good in the long term. An example problem which results is that of **fragmentation**. This is caused by the fact a given sequence of calls to `malloc` and `free` might leave small gaps in the heap which are not big enough to be used by subsequent calls to `malloc`. These small gaps represent wasted memory; sometimes we might try to compact the heap to remove the gaps but this is typically expensive. Thus there is already a clear trade-off between performance and memory footprint.

```
1 typedef struct __meta
2 {
3   struct __meta* prev;
4   struct __meta* next;
5
6           int     size;
7 }
8 meta;
```

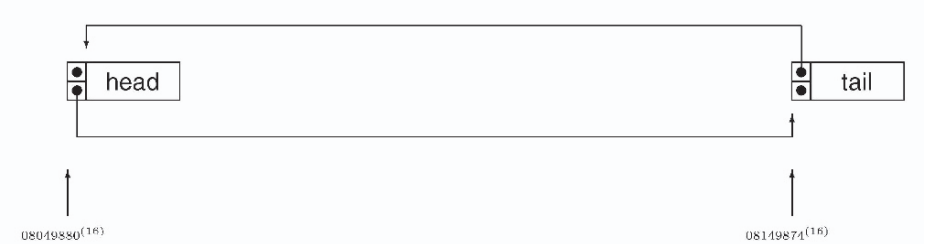
Listing 10.3 A C code fragment which describes support structures for the example linked-list-based heap implementation.

- The trade-off described above is made more complicated by the fact that calls to `malloc` and `free` themselves represent a performance overhead. Such calls can occur frequently in the program and so overhead relating to their execution should ideally be kept to a minimum. The implication for this is that although very effective algorithms for allocation (e.g., to minimise memory footprint) might exist, their *effectiveness* needs to be balanced against their *efficiency*.
- As in our example, imagine a program calls `malloc` to allocate two blocks of memory for arrays `A` and `B`. Careful management of allocation is crucial to program correctness: the library must be careful that the blocks it allocates for `A` and `B` do not overlap for example. If this were to happen, the program will almost certainly behave incorrectly since writing into an element of `A` could destroy an overlapping element of `B` and vice versa.

We can give a much more practical description of how a simple implementation of `malloc` and `free` might work by actually presenting one ! Of course there are many ways to approach the problem of heap management; we take a basic approach by modelling the heap as a (doubly) linked-list where each node in the list represents an allocated block. The idea is that each node is represented by two parts: a structure detailed in Listing 10.3 which describes meta-data such as the linked-list pointers (`prev` and `next`) and the block size (`size`), followed by the actual block of memory allocated to the program.

We perform allocation within a global array (which represents the memory associated with the dynamic data section) called `heap_data` shown in Listing 10.4; this array has a maximum size which represents a limit on the heap size. To initialise the heap, we use the `heap_init` function which constructs two meta-data structures at the start and end of `heap_data` called `heap_head` and `heap_tail`. To show what is going on, Figure 10.3 and Figure 10.4 show a step-by-step example use of the heap. After initialisation, Figure 10.3a shows the linked-list structure consisting of two nodes (i.e., `heap_head` and `heap_tail`) which span a large expanse of memory in which the heap can grow.

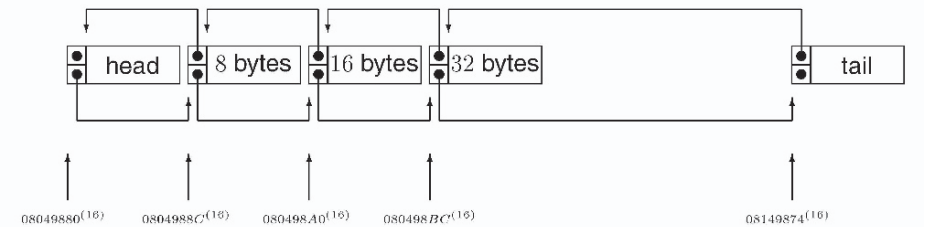
After initialisation, we can start using the heap via `malloc` and `free` style functions which, to avoid clashes, we call `heap_malloc` and `heap_free` and detail in Listing 10.5. Essentially, both functions manage the linked-list structure already in place. When the program wants to allocate some memory via a call to `heap_malloc`, the function traverses the linked-list starting at `heap_head`. At



(a) Initial heap state.

(b) After execution of `a = heap_malloc( 8 );`.

(c) After execution of `b = heap_malloc( 16 );`.



(d) After execution of `c = heap_malloc( 32 );`.

Figure 10.3 Example use of a linked-list-based heap implementation.

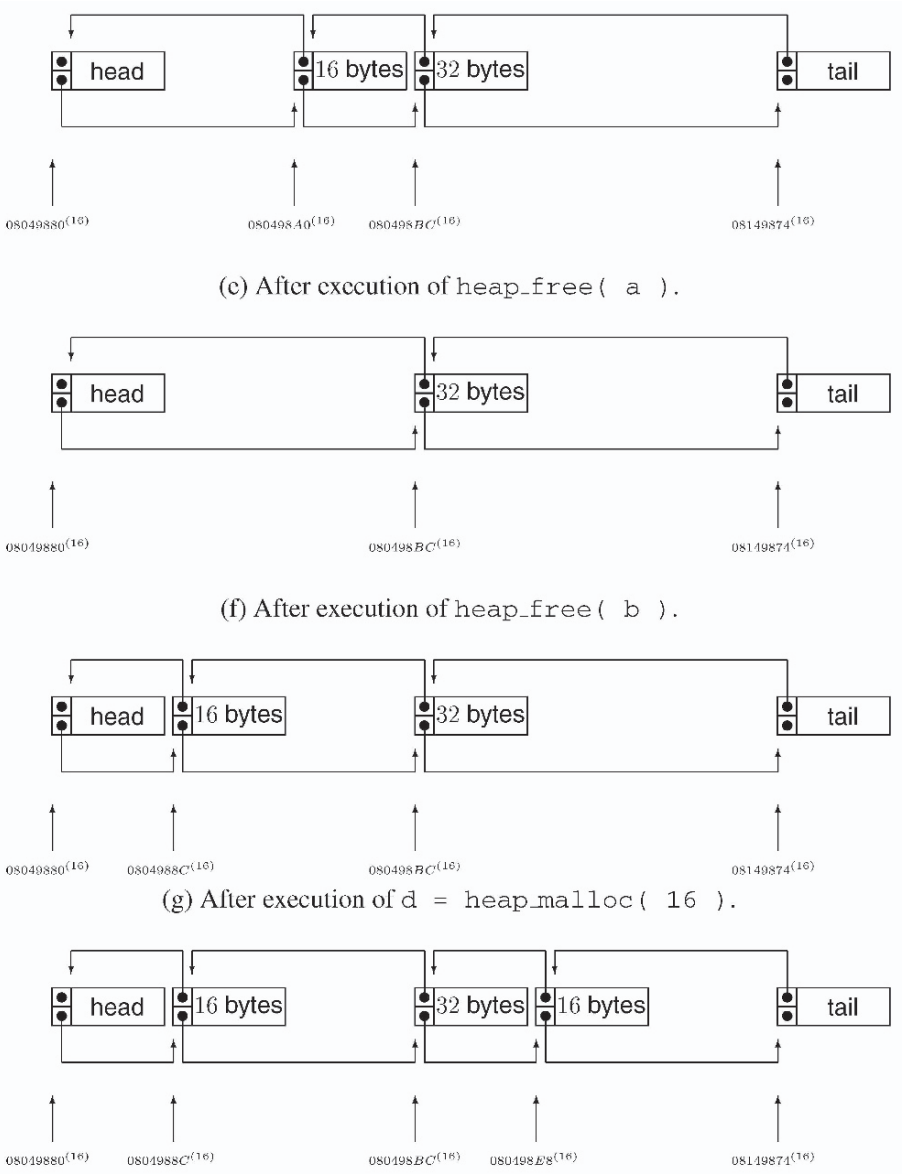


Figure 10.4 Example use of a linked-list-based heap implementation.

```
uint8 heap_data[ HEAP_SIZE ];

meta* heap_head;
meta* heap_tail;

void heap_init()
{
  heap_head = ( meta* )( heap_data                                  );
  heap_tail = ( meta* )( heap_data + HEAP_SIZE - sizeof( meta ) );

  heap_head->prev =      NULL;
  heap_head->next = heap_tail;
  heap_head->size =         0;

  heap_tail->next =      NULL;
  heap_tail->prev = heap_head;
  heap_tail->size =         0;
}
```

Listing 10.4 A C code fragment which initialises the example linked-list-based heap implementation.

each step, it checks the gap between the current node and the next node: if there is enough space to allocate the requested memory, then the linked-list is updated to include a new node and a pointer to the associated block is returned for use by the program. When the program want to deallocate some memory via a call to `heap_free` roughly the opposite occurs: the function removes the associated node from the linked-list so that the associated memory is available for subsequent allocations.

Figure 10.3a-d shows the state of the linked-list as three blocks of memory are allocated; since there is no reason to do otherwise, the blocks are allocated contiguously starting at `heap_head` so that the heap grows upward in memory. Figure 10.4e-f show the state of the linked-list after two of the three blocks are deallocated. The list structure remains in place but after deallocation there is a gap at the start of the heap. Finally Figure 10.4g-h shows two further blocks are allocated: the first fits into the gap left as a result of the previous deallocations, the second cannot fit into the (still smaller) gap that remains, and so is allocated at the end of the heap growing it further upward. Notice that there is now a slight fragmentation problem: at the start of the heap we now have a small gap which is quite unlikely to be useful and hence represents wasted memory.

10.3 Executable Versus Object Files

Most components of the tool-chain work with two types of files. A rough definition of the difference between the two types is that an **executable file** is ready for execution by the processor, an **object file** is in a form ready for further processing by the tool-chain; sometimes these are termed **load modules** and **object modules**, we term the content of an executable file a **program image**. Typically an executable file con-

```
1  void* heap_malloc( int n )
2  {
3    for( meta* ep = heap_head; ep != NULL; ep = ep->next )
4    {
5      meta* pp = ep->prev;
6      meta* np = ep->next;
7
8      uint8* ep_beg = ( uint8* )( ep );
9      uint8* ep_end = ( uint8* )( ep ) + sizeof( meta ) + ep->size;
10     uint8* np_beg = ( uint8* )( np );
11     uint8* np_end = ( uint8* )( np ) + sizeof( meta ) + np->size;
12
13     uint32 gap = np_beg - ep_end;
14
15     if( gap >= ( sizeof( meta ) + n ) )
16     {
17       meta* p = ( meta* )( ep_end );
18
19       ep->next =  p;
20        p->prev = ep;
21
22       np->prev =  p;
23        p->next = np;
24
25        p->size =  n;
26
27       return ( void* )( ( uint8* )( p ) + sizeof( meta ) );
28     }
29   }
30
31   return NULL;
32 }
33
34 void heap_free( void* p )
35 {
36   meta* ep = ( meta* )( ( uint8* )( p ) - sizeof( meta ) );
37
38   printf( "freeing %p with entry at %p\n", p, ep );
39
40   meta* pp = ep->prev;
41   meta* np = ep->next;
42
43   pp->next = np;
44   np->prev = pp;
45 }
```

Listing 10.5 A C code fragment which implements `malloc` and `free` style functions in the example linked-list-based heap implementation.

tains only machine code whereas an object file contains machine code along with **meta-data** which details features in the machine code. As an analogy, one might equate executable and object files to plain text files versus those produced by a word processor: the latter often include rich sets of meta-data to specify formatting and layout while the former includes only the raw text.

John Levine [37] gives an excellent overview of various executable and object file formats. Although the Executable and Linking Format (ELF) is perhaps more state of the art, to simplify our presentation we consider the `a.out` format which first appeared in early versions of UNIX. It exists in many flavours but we will consider a somewhat typical example of relocatable `a.out` which allows us to describe both

```
struct header
{
  uint32 magic;           // magic number
  uint32 text_size;       // section size of text
  uint32 data_size_i;     // section size of data (  initialised)
  uint32 data_size_u;     // section size of data (uninitialised)
  uint32 stab_size;       // section size of symbol table
  uint32 entry;           // entry point
  uint32 text_reloc_size; // section size of text relocation
  uint32 data_reloc_size; // section size of data relocation
}
```

Listing 10.6 The `a.out` header format.

```
struct reloc_entry
{
  uint32 address;     // offset of address that needs relocation
  uint32 symbol : 24, // offset into symbol table
         rel_pc :  1, // address is relative to program counter
         length :  2, // length of address
         extern :  1, // address is external
         rel_go :  1, // address is relative to global offset
         rel_jt :  1, // address is relative to jump table
         rel_ep :  1, // address is relative to entry point
         copy   :  1; // address should be copied from symbol
}
```

Listing 10.7 The `a.out` relocation entry format.

executable and object files. One of the major advantages of `a.out` is simplicity. An executable or object file is described by seven sections; throughout our discussion of these sections, any field which describes a size is measured in bytes. The header section, described by the C structure in Listing 10.6, allows quick analysis of the content: it includes the `magic` field which identifies the file type, fields such as `text_size` that specify the size of other sections in the file, and the `entry` field which specifies the entry point (i.e., for an executable `a.out` file, the address execution should start at). The text section contains the machine code instructions which constitute the program, the data section contains initialised data; in an executable `a.out` file these sections are used to populate the text and initialised data areas in the process address space.

Following the text and data sections are two sections that specify their respective **relocation entries**. Relocation entries have a standard format which is described by the C structure in Listing 10.7. In terms of the processor, relocation entries relate to addresses (or pointers) within the text or data sections which need attention when two object files are linked together or an executable file is loaded into memory. The `address` field contains an offset, into either the text or data section, of the item which needs attention; the `symbol` field specifies the index in symbol table in which information about the item in question is stored. Then follows a number of flags which specify more detailed information, the most important are the `rel_pc` flag that specifies whether the item is updated relative to the *PC* or in an absolute

```
struct stab_entry
{
  uint32  name; // offset into string table for symbol name
  uint8   type; // symbol type
  uint8  other;
  uint16 debug;
  uint32 value; // symbol value
}
```

Listing 10.8 The `a.out` symbol table entry format.

manner; the `length` flag that specifies the length of the item being updated; and the `external` flag which is set if the item relates to an item outside the current object file (if it is unset, then the item is local to the current file).

As programmers, we typically use human readable names to refer to the addresses that relocation entries identify; we call such names **symbols**, essentially a symbol maps a name to an address. The string table section is a number of null-terminated strings which are used to specify the names, or identifiers, of symbols. Each symbol is represented by an entry in the `a.out` symbol table; like a relocation entry, each symbol table entry has a standard format as shown in Listing 10.8. The `name` field contains an offset into the string table for the human readable symbol name. The `type` field determines the symbol type; for example, it might specify that the symbol exists in the text or data section or be a symbol that is specified but should remain untouched. The `other` and `debug` fields contain information used by debuggers and finally the `value` field notionally contains the value of the symbol; for symbols in the text and data sections this is an address.

10.4 Linkers

Imagine we take the C program represented by the source files P_{SRC} and Q_{SRC} and want to compile it. The first step is to compile the source files into object files which we will refer to as P_{OBJ} and Q_{OBJ} and whose content could roughly be described as follows. We expect that P_{OBJ} will

- have a text section corresponding to the instructions that implement the `main` function,
- have an uninitialised data section which is eight bytes in size and corresponds to the two pointers `A` and `B`,
- have an initialised data section which is 23 bytes in size and corresponds to the string `the dot product is %d\n`,
- have a symbol table which includes the local symbols `A`, `B`, `main` and the external symbol `dot`,
- have a text relocation section that includes an entry which notes that the symbol `dot` is external and that the call to `dot` needs relocation.

In contrast, we expect that Q_{OBJ} will

- have a text section corresponding to the instructions that implement the `dot` function,
- have uninitialised and initialised data section which are both empty,
- have a symbol table which includes the local symbol `dot`.

We can explore these descriptions by actually performing the compilation step for real using the GCC compiler. Imagine we save P_{SRC} and Q_{SRC} in the source files `P.c` and `Q.c` and then invoke GCC to compile them into the object files `P.o` and `Q.o`. From the UNIX command prompt this is achieved as follows:

```
bash$ ls
P.c  Q.c
bash$ gcc -std=gnu99 -c P.c
bash$ gcc -std=gnu99 -c Q.c
bash$ ls
P.c  P.o  Q.c  Q.o
```

We can verify our original descriptions by inspecting the object files `P.o` and `Q.o`. For example, the string tables in the resulting object files can be inspected using the `strings` command:

```
bash$ strings P.o
the dot product is %d
bash$ strings Q.o
```

The output shows that the string table in `P.o` has one entry which corresponds to the string we expected. Similarly, we can inspect the symbol tables using the `nm` command:

```
bash$ nm P.o
00000004 C A
00000004 C B
00000000 T main
         U dot
         U malloc
         U printf
         U atoi
bash$ nm Q.o
00000000 T dot
```

The output details the address of each symbol followed by the type (represented by a single character) and the name. For example, symbols `A` and `B` are both "common" symbols (denoted 'C') which will be placed in the uninitialised data section; symbols `dot` and `main` are both within the text sections of `P.o` and `Q.o` respectively (denoted 'T') but note that within `P.o` the symbol `dot` is "undefined" (denoted 'U') since it is defined externally (i.e., in `Q.o`); symbols `atoi`, `malloc` and `printf` are all "undefined" since they are library functions.

We can describe the two object files in a more easily digestible, tabular form in Figure 10.5. Now imagine we want to combine P_{OBJ} and Q_{OBJ} to produce a file

Symbol	Type	Address	External
`A`	Uninitialised Data	$00000004_{(16)}$	False
`B`	Uninitialised Data	$00000004_{(16)}$	False
`main`	Text	$00000000_{(16)}$	False
`dot`	Unknown		True
`malloc`	Unknown		True
`printf`	Unknown		True
`atoi`	Unknown		True

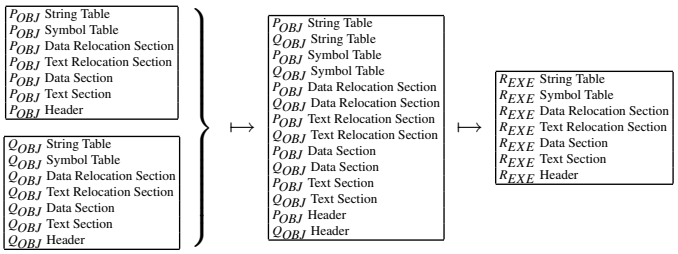
(a) Symbol table for P_{OBJ}.

Symbol	Type	Address	External
`dot`	Text	$00000000_{(16)}$	False

(b) Symbol table for Q_{OBJ}.

Figure 10.5 A tabular description of the symbol tables for the object files P_{OBJ} and Q_{OBJ}.

R_{EXE} which we can execute. This combination is performed by the linker and (very) roughly consists of two steps described diagrammatically (if a little too abstractly) below:



Firstly the linker performs a layout step whereby the sections from P_{OBJ} and Q_{OBJ} are merged together and placed within a partly finished version of R_{EXE}. Note that each section, which would have had a known offset within either P_{OBJ} or Q_{OBJ}, has been moved so that it now has a new offset within R_{EXE}. Secondly the linker manipulates the content of each section so they make sense within the new executable file. For example since we have assigned them new offsets, any references to items in the symbol tables for P_{OBJ} or Q_{OBJ} need to be adjusted. This second step is the most involved: it demands that we consider and solve four main problems which we expand upon below.

Symbol	Type	Address	External
`A`	Uninitialised Data	$0804971C_{(16)}$	False
`B`	Uninitialised Data	$08049720_{(16)}$	False
`main`	Text	$080483AC_{(16)}$	False
`dot`	Text	$080484C4_{(16)}$	False
`malloc`	Unknown		True
`printf`	Unknown		True
`atoi`	Unknown		True
`_start`	Text	$08048300_{(16)}$	False

(a) Symbol table for the dynamically linked R_{EXE}.

Symbol	Type	Address	External
`A`	Uninitialised Data	$080A39F0_{(16)}$	False
`B`	Uninitialised Data	$080A39F4_{(16)}$	False
`main`	Text	$080481F0_{(16)}$	False
`dot`	Text	$08048308_{(16)}$	False
`malloc`	Text	$0804A1D8_{(16)}$	False
`printf`	Text	$080492AC_{(16)}$	False
`atoi`	Text	$080487F0_{(16)}$	False
`_start`	Text	$08048100_{(16)}$	False

(b) Symbol table for the statically linked R_{EXE}.

Figure 10.6 A tabular description of the symbol tables for the dynamically and statically linked executable R_{EXE}.

10.4.1 Static and Dynamic Linkage

We already described library functions as being a collection of commonly used functions which can be linked into an executable file by the linker; the idea is essentially that the functions only need exist in one place and are hence more maintainable. In reality, we need to be more exact about how the linker combines a library with existing object files in order to create an executable file. That is, there are two options for dealing with the "unknown" symbols in Figure 10.5 that represent the library functions `malloc`, `printf` and `atoi`:

Dynamic Linking In this case, the operating system maintains one *single* copy of any given library in memory. If an executable file is loaded and is the first to use the library (i.e., no other loaded executables use it), then the operating system also has to load the library, otherwise it does not; the library is already resident in memory. When the loader loads the executable file into memory, it patches references to library functions so they refer to this single instance.

The advantages of this approach are that memory is conserved and the time spent loading libraries is potentially reduced. The disadvantages are that the loader needs to be more complicated, and we need to be very careful that exactly the right library is available and used (i.e., there is not a mismatch between functions the executable file wants to use and those provided in the library loaded by the operating system).

Static Linking In this case, the linker takes all the library functions and combines them directly into *every* executable file. There is no need for the loader to take any remedial action when the executable file is loaded into memory since references to the library functions are all internal.

The advantages of this are the lack of problems with respect to incompatibilities between the executable file and available libraries: the executable file contains all the machine code needed. The disadvantages are that now we might find many processes in memory all with their own identical copies of a library linked into them; this could be viewed as somewhat wasteful.

We can see how the two options work by continuing the running example of using GCC to describe a real compilation process; again the end result is shown in a tabular form in Figure 10.6. We can invoke the linker, and instruct it to use dynamic linking, using another call to GCC:

```
bash$ gcc -std=gnu99 -dynamic -o R P.o Q.o
bash$ ls
P.c  P.o  Q.c  Q.o  R
```

Actually dynamic linkage is the default so this command would have worked the same way without the extra option. Either way, we can again use `strings` and `nm` to inspect the string and symbol tables:

```
bash$ strings R
the dot product is %d
...
bash$ nm R
0804971c B A
08049720 B B
080483ac T main
080484c4 T dot
         U malloc@@GLIBC_2.0
         U printf@@GLIBC_2.0
         U atoi@@GLIBC_2.0
08048300 T _start
...
```

This time, the output has been truncated and massaged a little for clarity. The string table is not too interesting: it contains the original string from `P.o` as we would expect, plus some extra strings that relate to functions that have been called and so on. The symbol table shows four more interesting features:

- the symbols `A` and `B` have been explicitly placed in the uninitialised data section,

- the symbols `dot` and `main` have been combined into a single text section in the executable file,
- the symbols `malloc`, `printf` and `atoi` are still marked "undefined" but have had an extra identifier tagged onto the end of their name to show which library (and version) needs to be used,
- a new symbol called `_start` has been introduced.

Now consider the same process except that we instruct GCC to perform static linking instead:

```
bash$ gcc -std=gnu99 -static -o R P.o Q.o
bash$ ls
P.c  P.o  Q.c  Q.o  R
bash$ strings R
the dot product is %d
malloc: using debugging hooks
realloc(): invalid pointer %p!
malloc: top chunk is corrupt
free(): invalid pointer %p!
...
bash$ nm R
080a39f0 B A
080a39f4 B B
080481f0 T main
08048308 T dot
0804a1d8 W malloc
080492ac T printf
080487f0 T atoi
08048100 T _start
...
```

This time, the output from `strings` and `nm` is significantly more verbose; in comparison to the dynamic case, there are two interesting features:

- the string table include strings defined within the library functions we have included (e.g., strings which relate to error messages in `malloc` and `free`) as well as our own,
- whereas symbols relating to `malloc`, `printf` and `atoi` library functions were previously "undefined", they now have addresses and are marked as being included in the text section.

10.4.2 Boot-strap Functions

In order to produce a genuinely executable file, the linker has to include some standard interface between a program and the operating system it executes within. In Figure 10.6 we can see new symbols that relate to new functions within the text section of the executable file: `_start` represents a function added by the linker to provide said interface. You can think of it as a library function that is automatically

included by the linker rather than included as a product of how the program was written or how GCC was invoked.

More specifically, `_start` acts to initialise aspects of the program when executed. For example, the `_start` function acts as a standard entry point for the executable file; this will be marked in the `a.out` header as where execution should begin. It is tasked with calling the `main` function by constructing and passing in the `argc` and `argv` arguments, and passing back the return value to the operating system.

10.4.3 Symbol Relocation

For the sake of simplicity, imagine the size of the text sections in P_{OBJ} and Q_{OBJ} are p and q bytes respectively, and that they were both originally generated assuming they would be loaded at address $00000000_{(16)}$ ready for execution. We can see this in Figure 10.5 where each of `main` and `dot` has address $00000000_{(16)}$ in P_{OBJ} and Q_{OBJ} respectively. To construct the executable file R_{EXE}, during the layout step we placed the two text sections directly after each other. As such, when the new executable is loaded into memory ready for execution what was the text section for P_{OBJ} will occupy addresses in the range $0 \ldots p-1$ while what was the text section for Q_{OBJ} will occupy addresses in the range $p \ldots p+q-1$.

This, of course, presents a problem: we have **relocated** what was the text section for Q_{OBJ} relative to where it thinks it should be; any absolute branch, for example, will be to the wrong address ! For example, since the text section for Q_{OBJ} has been relocated to address p, if Q_{OBJ} included an absolute branch to address x, then this branch needs to be patched so it directs control-flow to address $p+x$ instead. A similar situation exists with the symbols `A` and `B` which do not yet have concrete addresses in P_{OBJ} or Q_{OBJ}. When the linker performs layout of the uninitialised data section, it assigns them concrete addresses that we can see in Figure 10.6. In the same way as the branch, each reference to `A` or `B` needs to be patched to cope with the fact that they have been relocated.

Without going into the details of how machine code instructions are actually patched, the process which solves this problem is easy to describe. Essentially, the linker examines each original object file in turn; for each symbol it updates the new symbol table (i.e., the one in R_{EXE}) so that the address matches the relocated address. Put more simply, in our example this means that in the symbol tables for object files P_{OBJ} and Q_{OBJ} have `main` and `dot` both at address zero; in the symbol table for the executable file their addresses are zero and p. Then, it looks for references (marked in the relocation table) within the object files to each symbol in the new symbol table. For each reference it identifies the associated instruction in the new text section, and patches it by setting the address to where the symbol was relocated to (i.e., the address in the new symbol table).

This problem can be (partly) mitigated by writing so-called **Position Independent Code (PIC)**. The rough idea is that *no* absolute addresses should exist within

the machine code; it can therefore be placed anywhere in memory and still work without any special treatment by the linker or loader. The means by which this is accomplished are somewhat involved: although some absolute addresses can be removed by using *PC*-relative branches for example, others are more problematic. For example, references to absolute addresses that represent data items might need to be routed through a table which houses the real address and is updated to suit the address at which the machine code is loaded. The typical trade-off is therefore that a program using PIC will be slightly larger and execute slightly more slowly than one which does not.

10.4.4 Symbol Resolution

The text section for P_{OBJ} included a reference to the `dot` symbol which is actually a function located in Q_{OBJ}. During the compilation step, GCC was happy to simply mark `dot` as an external symbol in P_{OBJ}; this is clear in Listing 10.5 where the symbol table for P_{OBJ} lists `dot` as an external symbol without a concrete address. However, during the linking step we need to patch instructions that refer to `dot` so that they are **resolved** to the concrete address of `dot` in the executable file, i.e., where `dot` was placed in the new text section of R_{EXE}.

Again ignoring how instructions are actually patched, the resolution process is similar to relocation. The linker again examines each original object file in turn; for each symbol marked as external, it processes the references to that symbol. Each reference details an instruction that needs to be patched so the previously missing address is filled in with the concrete address of the symbol within the new executable file.

10.5 Assemblers

Above the linker is the **assembler** tool which is tasked with translation of low-level **assembly language** programs either directly into an executable file or into an object file which is converted into an executable form later. Statements in an assembly language program correspond almost one-to-one with the machine instructions and so in contrast to portable, high-level languages such as C, most assembly languages are closely tied to the processor ISA. Example assemblers include `gas`, which forms part of the GCC tool-chain. Systems such as C−− [52] enhance the basic remit of assemblers by adding selective higher-level features to the assembly process, and also by attempting to offer some degree of portability.

In this section we try to describe assemblers, assembly language and assembly language programming in an ISA agnostic way with MIPS32 as a concrete example. That is, the concepts introduced use MIPS32 for specific examples but are easily translated to other assembly language formats and processor ISAs. Further, we shy

away from matters relating to executing assembly language programs using a physical processor or simulator; instead we provide a dedicated introduction to SPIM [35] in Appendix A.

10.5.1 Basic Assembly Language Statements

With some exceptions, assembly languages are typically line-based: each line of the assembly language program contains a statement which can be dealt with by the assembler more or less independently from the other statements. Two other distinguishing features are common to most assembly languages, namely that the languages are typeless and unstructured. In high-level languages symbols have a type, for example a variable in C might be declared as being an `int` or a `char` or a `float`; an assembly language works at the machine level and so the only real type is a machine word. In high-level languages control-flow statements such as loops and conditionals enforce a structure to the program; an assembly language program has no such structure with the only control-flow being via individual jumps and branches. These features (and others) mean that assembly language programming is a challenge. The normal philosophy of assembler design is that right or wrong, they do exactly what you tell them; although they will check for syntax errors within the input program, the assembler will happily let you write programs which are dubious in terms of type errors and so on. The main benefit of an assembler is thus automation of tasks that would normally have to be performed by hand.

The operation of an assembler can roughly be described as reading each line, parsing fields of constituent statements and constructing equivalent machine code encodings. The assembler keeps an **address counter** as it processes assembly language statements: each statement that causes the assembler to produce some machine code output also causes the address counter to be incremented. The statements on each line are drawn from three classes: **machine instructions**, assembler **aliases** or **pseudo-instructions**, and assembler **directives**. Although we discuss their meaning and use later, all are specified using roughly the same general format:

```
[<label>:] <mnemonic> [<operand> ...] [#<comment>]
```

Note that the fields in square brackets are optional; the meaning of each field demands further explanation:

Label The optional label associates an address with the related statement; it basically specifies that the label value should be set equal to the current value of the address counter. In the terminology we already introduced while discussing executable and object files, a label defines a symbol: the two things are basically the same, but in the context of assemblers and assembly language the term label is more common (so we stick to it here).
Thus, instead of writing an instruction to jump to address zero we might instead write an instruction that jumps to label `f` and let the assembler work out the address `f` is located at; instead of writing an instruction to load a value from address

eight we might write an instruction that loads the value associated with the label `a`. Note that because it would confuse the process of parsing the statement, labels cannot be the same as reserved words (which include statement mnemonics). For example, you cannot specify a label called `syscall` since this is also a MIPS32 instruction mnemonic.

Mnemonic The statement mnemonic is the keyword which determines the action to be performed: it plays the role of the operator or opcode when the statement specifies a machine instruction, essentially allowing the programmer to write a human-readable name instead of the numeric opcode. For example, the statement

```
add $2, $3, $4
```

specifies an instruction to perform addition: use of the mnemonic `add` instead of the numeric constants $000000_{(2)}$ and $100000_{(2)}$ (which form the `opcode` and `funct` fields for an addition instruction) clearly makes life much easier for the programmer ! Since some statements have no operands and both labels and comments are optional fields, the mnemonic is the only compulsory part of a statement.

Operand An operand is an item which is operated on in some way by a statement; source operands are used as input to the statement, target operands are produced as output. Operands within an assembly language statement allow the programmer to specify what these items are and where they are located. The number of operands depends on the type of statement. For example, the `add` instruction requires three operands: two source operands, which are located in general-purpose registers, and one target operand which will be written into a general-purpose register. As such, writing the statement

```
add $2, $3, $4
```

specifies an `add` instruction via the mnemonic, one target operand in the form of `$2` or *GPR*[2] and two source operands in the form of `$3` and `$4` or *GPR*[3] and *GPR*[4]. As well as register operands, immediate values can be specified in either decimal or hexadecimal form. As such, the statements

```
addi $2, $2, 10
```

and

```
addi $2, $2, 0xA
```

both specify instructions which increment *GPR*[2] by the constant value ten. Decimal constants can be positive or negative, i.e., specified using standard notation as `10` or `-10`, but hexadecimal constants are interpreted directly as the corresponding binary vector. For example the constant value `-0xA` is invalid. Finally, labels can be included as operands in certain statements. For example, one might write

```
beq $2, $3, f
```

to specify an instruction which branches to the address associated with the address `f` if *GPR*[2] and *GPR*[3] contain the same value. The interpretation of labels is dependent on the statement type; we will examine this issue in more detail later.

Comment The role of comments in assembly language programs is the same as in higher-level programs: they allow the programmer to annotate a program with human-readable information which aids understanding. Although the comment field is optional, commenting assembly language programs is often regarded as *even more* important than in higher-level programs because said programs are usually much harder to understand without comments. A comment starts with the hash character and extends to the end of the line; content after the hash character is discarded by the assembler before further processing so that, for example, the statement

```
add $2, $3, $4 # this is a comment
```

is equivalent to the statement

```
add $2, $3, $4
```

since the comment field is simply discarded.

Some standard lexicographical conventions found within other programming languages are also found within assembly language. For example, white space (includes tabs) is not significant except where used as a divider between fields in a statement. This means that the statement

```
add $2, $3, $4
```

can also be written as

```
add $2, $3,$4
```

or

```
add $2 , $3 , $4
```

but not as

```
add$2,$3,$4
```

or

```
add $2 $3 $4
```

Although blank lines and lines containing only a comment are permitted, they are ignored. For example, the statement

```
# this is a comment
```

is equivalent to a blank line and hence ignored, as is an entirely blank line.

10.5.2 Using Machine Instructions

Table 10.1 gives a brief overview of assembly language statements that translate into machine instructions for the reduced MIPS32 ISA introduced in Chapter 5; the left-hand column gives the statement format and the right-hand column gives

Statement	Meaning
`add rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] + GPR[rt])$
`sub rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] - GPR[rt])$
`mul rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \cdot GPR[rt])$
`mult rs, rt`	$(HI, LO) \leftarrow (GPR[rs] \cdot GPR[rt])$
`and rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \wedge GPR[rt])$
`or rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \vee GPR[rt])$
`xor rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \oplus GPR[rt])$
`nor rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \bar{\vee} GPR[rt])$
`sllv rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \ll GPR[rt]_{4\ldots0})$
`srlv rd, rs, rt`	$GPR[rd] \leftarrow (GPR[rs] \gg GPR[rt]_{4\ldots0})$
`addi rt, rs, imm`	$GPR[rt] \leftarrow (GPR[rs] + ext(imm))$
`mfhi rd`	$GPR[rd] = HI$
`mflo rd`	$GPR[rd] = LO$
`mfc0 rt, rd`	$GPR[rt] = CP0[rd]$
`mthi rs`	$HI = GPR[rs]$
`mtlo rs`	$LO = GPR[rs]$
`mtc0 rt, rd`	$CP0[rd] = GPR[rt]$
`lw rt, imm(rs)`	$GPR[rt] \leftarrow MEM[GPR[rs] + ext(imm)]$
`sw rt, imm(rs)`	$MEM[GPR[rs] + ext(imm)] \leftarrow GPR[rt]$
`beq rs, rt, off`	if $GPR[rs] = GPR[rt], PC \leftarrow PC + ext(off : 00_{(2)})$
`bne rs, rt, off`	if $GPR[rs] \neq GPR[rt], PC \leftarrow PC + ext(off : 00_{(2)})$
`blez rs, off`	if $GPR[rs] \leq 0, PC \leftarrow PC + ext(off : 00_{(2)})$
`bgtz rs, off`	if $GPR[rs] > 0, PC \leftarrow PC + ext(off : 00_{(2)})$
`bltz rs, off`	if $GPR[rs] < 0, PC \leftarrow PC + ext(off : 00_{(2)})$
`bgez rs, off`	if $GPR[rs] \geq 0, PC \leftarrow PC + ext(off : 00_{(2)})$
`b off`	$PC \leftarrow PC + ext(off : 00_{(2)})$
`j imm`	$PC \leftarrow (imm : 00_{(2)})$
`jr rs`	$PC \leftarrow GPR[rs]$
`jal imm`	$GPR[31] \leftarrow PC + 8, PC \leftarrow (imm : 00_{(2)})$
`nop`	no operation
`syscall`	cause system call exception
`eret`	return from exception

Table 10.1 A tabular description of an assembly language for a reduced MIPS32 ISA.

the meaning in terms of the processor state. The instructions are roughly grouped into four classes: those which perform arithmetic, for example the addition or `add` instruction; those which perform access to memory, for example the load word or `lw` instruction; those which alter the flow of control, for example the unconditional branch or `b` instruction; and those which are deemed special in some sense, for example the system call or `syscall` instruction.

10.5.2.1 Arithmetic Instructions

Arithmetic instructions which manipulate values held in the general-purpose registers typically form the majority of any program. We already introduced a few examples of this type without explaining them; for example we have already written the statement

```
add $2, $3, $4
```

to specify an `add` instruction which takes $GPR[3]$ and $GPR[4]$, adds them together and writes the result back into $GPR[2]$. It is worth commenting immediately on the usefulness of having $GPR[0] = 0$. Because of this fact, we can set registers to any (small) constant using `addi`. To see how, consider the instruction:

```
addi $2, $0, 10 # set GPR[2] = GPR[0] + 10 = 0 + 10 = 10
```

There are no great mysteries involved in what the assembler does with such an instruction. It parses the statement, looks up what the encoding should be for this machine instruction, then produces the resulting encoded instruction as output. In this case, the instruction semantics

$$GPR[rt] \leftarrow GPR[rs] + ext(imm)$$

are filled in to read

$$GPR[2] \leftarrow GPR[0] + 10.$$

This is translated by the assembler into the 32-bit machine code output

$$00100000000000100000000000001010_{(2)}.$$

The most-significant six bits are $001000_{(2)}$ which mean the opcode implies an `addi` or add immediate instruction. The next two sets of five bits specify the source and target registers: in this case we read from $GPR[0]$ and write into $GPR[2]$ so the values are $00000_{(2)}$ and $00010_{(2)}$ respectively. The final sixteen bits are the immediate value $0000000000001010_{(2)} = 10_{(10)}$. Of course, this works with negative constants as well; we just need to be careful to deal with the immediate value in the correct way. That is we need to ensure the sign, in terms of twos-complement representation, is correct. For example, consider encoding the instruction `addi $2, $0, -10` instead. This results in the output

$$00100000000000101111111111110110_{(2)}$$

where everything is the same except the immediate value $1111111111110110_{(2)}$ which is the 16-bit twos-complement representation of $-10_{(10)}$.

The reduced MIPS32 ISA described in Table 10.1 does not include an instruction for division. So, as a limited study of using the suite of arithmetic instructions, consider the problem of taking an integer value x and dividing it by the constant 3 to give an integer result, i.e., computing $q = \lfloor x/3 \rfloor$, or more generally $q = \lfloor x/d \rfloor$ for some d. To avoid the actual division, the trick is to estimate $1/d$, the reciprocal of

Input: An integer value x.
Output: The integer value $q = \lfloor x/3 \rfloor$.

```
1 DIVIDE-BY-3(x)
2     q ← (x ≫ 2) + (x ≫ 4)
3     q ← (q ≫ 4) + q
4     q ← (q ≫ 8) + q
5     q ← (q ≫ 16) + q
6     r ← x − (3 · q)
7     return q + ((11 · r) ≫ 5)
8 end
```

Algorithm 10.1: An algorithm for performing integer division by three.

```
1  .globl main
2
3  main:  addi    $3, $0, 94 # set x to 94
4         addi    $5, $0,  2
5         srlv    $6, $3, $5
6         addi    $5, $0,  4
7         srlv    $7, $3, $5
8         add     $4, $6, $7 # set q to ( x >>  2 ) + ( x >> 4 )
9         addi    $5, $0,  4
10        srlv    $6, $4, $5
11        add     $4, $6, $4 # set q to ( q >>  4 ) + q
12        addi    $5, $0,  8
13        srlv    $6, $4, $5
14        add     $4, $6, $4 # set q to ( q >>  8 ) + q
15        addi    $5, $0, 16
16        srlv    $6, $4, $5
17        add     $4, $6, $4 # set q to ( q >> 16 ) + q
18        addi    $5, $0,  3
19        mul     $6, $4, $5
20        sub     $6, $3, $6 # set r to x - ( 3 * q )
21        addi    $5, $0, 11
22        mul     $6, $6, $5
23        addi    $5, $0,  5
24        srlv    $6, $6, $5
25        add     $4, $4, $6 # set q to q + ( ( 11 * r ) >> 5 );
26        addi    $2, $0, 10
27        syscall            # exit
```

Listing 10.9 MIPS32 implementation of integer division by three.

d, and then multiply it with x to get the result. For the case of $d = 3$ we find that $1/3 \simeq 0.0101...01_{(2)}$. More simply this can be written

$$1/3 \simeq 1/2^2 + 1/2^4 + 1/2^6 + \cdots + 1/2^{30} + \cdots$$

so that the division $x/3$ can be written

$$x/3 \simeq x/2^2 + x/2^4 + x/2^6 + \cdots + x/2^{30} + \cdots .$$

This looks unpleasant, but actually it is not too bad because we already know that to compute $x/2^k$ we simply need to shift x right k bits. There are some subtleties with this however, mainly in terms of accuracy. Specifically, we are limited to 32-bit operations whereas the reciprocal is infinitely long and we are throwing away some of it with each right-shift of x we perform. Thus, some remedial operations are required which we will not go into here; for more detail on the method, see the marvellous discussion of this topic by Henry Warren [64, Chapter 10].

A basic algorithm for performing the division is shown in Algorithm 10.1; note that we could eliminate some of the multiplications by constants, but these are left in place to demonstrate the `mul` instruction. We can implement the algorithm using MIPS32 machine instructions as demonstrated in Listing 10.9; since this is our first program of any length, the features in it demand some further discussion. Firstly, note that the program is straight-line: it contains no branches so execution begins at the top and ends at the bottom. Usually the first thing to think about when writing such a program is the **register allocation**, i.e., which variables in the algorithm will be held in which machine registers. In this program we use register $GPR[3]$ or `$3` to hold the value of x; register $GPR[4]$ or `$4` to hold the accumulated value of q; and registers $GPR[5]$, $GPR[6]$ and $GPR[7]$ or `$5`, `$6` and `$7` to hold temporary or intermediate values.

Once we know which register hold which variables, things start to be easier to understand. The second thing to think about is the decomposition of the algorithm into small steps which map onto the operations we can perform using the instruction set. For example, the first instruction is an application of the discussion above about setting registers to an immediate value: we set $GPR[3]$ or x to the value we want to divide by, in this example $x = 94$ so we are expecting the result 31. Subsequent instructions implement (more or less) exactly the steps in the algorithm. For example, the next five machine instructions implement the step $q \leftarrow (x \gg 2) + (x \gg 4)$. Think about how this works: the first instruction sets $GPR[5]$ to the constant 2; the second instruction right-shifts the contents of $GPR[3]$, i.e., x, by the contents of $GPR[5]$, i.e., 2, and stores the result in $GPR[6]$; the third and fourth instructions perform a similar function except with a shift distance of 4; finally, the fifth instruction takes the intermediate results in $GPR[6]$ and $GPR[7]$, i.e., the results $x \gg 2$ and $x \gg 4$, adds them together and stores them in $GPR[4]$, i.e., q. The final two instructions in the program perform a system call that halts the program; this is useful if simulating with SPIM but otherwise nothing to worry about, we will cover this later.

None of these instructions are particularly interesting as far as the encoding done by the assembler, it simply fills in the relevant register operands for the appropriate encoding and produces the resulting output. Take the third, right-shift instruction which reads `srlv $6, $3, $5` for example. The semantics of this instruction are

$$GPR[rd] \leftarrow GPR[rs] \gg GPR[rt]$$

which for this specific example mean

$$GPR[6] \leftarrow GPR[3] \gg GPR[5].$$

The assembler selects the encoding for a right-shift instruction and produces the 32-bit machine code output

$$00000000011001010011000000000110_{(2)}.$$

The most-significant and least-significant six bits are $000000_{(2)}$ and $000110_{(2)}$ respectively; this means the opcode and ALU function fields are set such that the instruction represents an `srlv` or logical right-shift. The middle three sets of five bits specify the registers: in this case we read from *GPR*[3] and *GPR*[5] and write into *GPR*[6], the values are $00011_{(2)}$, $00101_{(2)}$ and $00110_{(2)}$ respectively.

This whole exercise might have seemed painful: it was a lot *more* effort than implementing the same algorithm in C for example. But notice that it was a lot *less* effort than implementing it in raw machine code: imagine the effort of encoding twenty or so instructions by hand, let alone ensuring the program they represent is correct !

10.5.2.2 Memory and Register Access Instructions

The most basic form of access to operands beyond the general-purpose register set is access to the special-purpose registers. Table 10.1 shows two main examples which are able to read and write values to and from the *HI* and *LO* registers, and to and from the co-processor register file *CP*0. The purpose of access to *HI* and *LO* is simple: `mult` is an arithmetic instruction which yields the 64-bit result of multiplying together two 32-bit operands. Since the ISA can only support 3-address style instructions with two source operands and one target operand, the instruction stores the multiplication result into *HI* and *LO*. These registers are not generally addressable so in order to perform further processing of the results we need to move them, using special instructions, into general-purpose registers. Therefore, a typically observed instruction sequence might be

```
mult $2, $3 # multiply together contents of GPR[2] and GPR[3]
mflo $4     # move low  32-bit part of result into GPR[4]
mfhi $5     # move high 32-bit part of result into GPR[5]
```

which performs the multiplication and then extracts the result parts in order to perform some further computation with them.

The second example of special-purpose register access is to the co-processor, and is more involved. Roughly speaking, this allows us to perform some special operations which we will revisit in Chapter 12.

Access to main memory via the load and store word instructions `lw` and `sw` is usually more often used than special-purpose register access. The format of these instructions is slightly different from the others in the sense that one specifies the base address and offset using a slightly odd, bracketed syntax. The base address is always a register operand; the offset is always a constant value, either a number or a label, written before the bracketed base address.

Consider the `lw` for example. The following example program attempts to load three values from addresses 0, 4 and 8 using a variety of offsets from a fixed base address of 4 held in $GPR[2]$:

```
addi $2, $0, 4  # set GPR[2] to 4
lw   $5, -4($2) # load MEM[GPR[2]-4] = MEM[4-4] = MEM[0] into GPR[5]
lw   $6,  0($2) # load MEM[GPR[2]+0] = MEM[4+0] = MEM[4] into GPR[6]
lw   $7,  4($2) # load MEM[GPR[2]+4] = MEM[4+4] = MEM[8] into GPR[7]
```

Encoding the memory access instructions follows the same pattern we have already seen; consider the first load as an example and remember that we need to be careful to specify the sign of the immediate value correctly as in the case of `addi`. The semantics for `lw` are

$$GPR[rt] \leftarrow MEM[GPR[rs] + ext(imm)]$$

which for this specific example is

$$GPR[5] \leftarrow MEM[GPR[2] - 4]$$

and is translated by the assembler into the 32-bit machine code output

$$10001100010001011111111111111100_{(2)}.$$

The most-significant six bits are $100011_{(2)}$ which mean the opcode implies an `lw` or load word instruction. The next two sets of five bits specify the source and target registers: in this case we read from $GPR[2]$ and write into $GPR[5]$ so the values are $00010_{(2)}$ and $00101_{(2)}$ respectively. The final sixteen bits are the immediate value $1111111111111100_{(2)}$ which is the 16-bit twos-complement representation of $-4_{(10)}$.

We could write the program above in a different way, loading the same three values using a variety of base addresses and a fixed offset:

```
addi $2, $0, -4 # set GPR[2] to -4
addi $3, $0,  0 # set GPR[3] to  0
addi $4, $0,  4 # set GPR[4] to  4
lw   $5,  4($2) # load MEM[GPR[2]+4] = MEM[-4+4] = MEM[0] into GPR[5]
lw   $6,  4($3) # load MEM[GPR[2]+4] = MEM[ 0+4] = MEM[4] into GPR[6]
lw   $7,  4($4) # load MEM[GPR[2]+4] = MEM[ 4+4] = MEM[8] into GPR[7]
```

At face value, this perhaps makes less sense. However, remember that we can replace the offset with a label. Imagine that the label `a` represents the address of an array of 32-bit integers in memory; rewriting the example as

```
addi $2, $0,  0 # set GPR[2] to 0
addi $3, $0,  4 # set GPR[3] to 4
addi $4, $0,  8 # set GPR[4] to 8
lw   $5,  a($2) # load MEM[0] = MEM[&a+0] ~ a[0] into GPR[5]
lw   $6,  a($3) # load MEM[4] = MEM[&a+4] ~ a[1] into GPR[6]
lw   $7,  a($4) # load MEM[8] = MEM[&a+8] ~ a[2] into GPR[7]
```

now makes more sense. Remembering that each element of the array is a 32-bit integer, i.e., a word or four bytes, the program now loads elements 0, 1 and 2 of the array into *GPR*[5], *GPR*[6] and *GPR*[7].

Of course, `sw` works in much the same way except we write values into memory rather than read values from it. As a brief example, consider swapping the first and second elements in the array `a` above:

```
addi $2, $0,  0 # set GPR[2] to 0
addi $3, $0,  4 # set GPR[3] to 4
lw   $5,  a($2) # load MEM[0] ~ a[0] into GPR[5]
lw   $6,  a($3) # load MEM[4] ~ a[1] into GPR[6]
sw   $6,  a($2) # store GPR[6] into MEM[0] ~ a[0]
sw   $5,  a($3) # store GPR[5] into MEM[4] ~ a[1]
```

However, beware of the requirements of both `lw` and `sw` to use word-aligned addresses, i.e., addresses which are a multiple of the word length, or more specifically a multiple of four bytes. Consider the short example program:

```
addi $2, $0, 1 # set GPR[2] to 1
lw   $3, 0($2) # attempt to load MEM[1] into GPR[3] causes exception
```

When the processor tries to execute the `lw` instruction an **error** or **exception** will occur because the address, i.e., $00000001_{(16)}$, is not word aligned. The implication of the exception is revisited in Chapter 12; for now, it is enough to think of it as crashing the program.

10.5.2.3 Control-Flow Instructions

Instructions for changing the flow of control can be roughly broken down using two characteristics as we discussed in Chapter 5: branches which are conditional (often termed jumps) versus those which are unconditional, and branches which use an offset versus an absolute address. The MIPS32 ISA contains branches with both characteristics, Table 10.1 includes a sub-set of them.

Consider a small example whereby we want to compute the maximum of two numbers x and y. To achieve this, we compute $x - y$. Then, if the result is negative (resp. zero), x must be less than (resp. equal to) y, whereas if the result is positive (resp. zero), x must be greater than (resp. equal to) y. Thus, we can alter the flow of control to store the maximum, i.e., which of x and y is the largest, based on the value $x - y$:

```
       addi $3, $0,  1 # set GPR[3] to 1, i.e. x = 1
       addi $4, $0,  2 # set GPR[4] to 2, i.e. y = 2
       sub  $2, $3, $4 # set GPR[2] to GPR[3] -  GPR[4]
       blez $2, take   # goto address "take" if GPR[2] <= 0
skip : add  $5, $0, $3 # set GPR[5] to GPR[3] + 0 = GPR[3]
       j    done       # goto address "done"
take : add  $5, $0, $4 # set GPR[5] to GPR[4] + 0 = GPR[4]
done : ...
```

Notice that we use two types of branch in this example and in both cases we are using labels to refer to addresses rather than numbers. Instruction #4 is a conditional

branch; if $GPR[2]$ is less than or equal to zero, then *PC* is set to the address `take`, i.e., the next instruction executed will be #7. Otherwise the *PC* is incremented as normal, i.e., the next instruction executed will be the #5. Instruction #6 is an unconditional branch: it always sets the *PC* to the address `done` so that the next instruction executed will be #8.

The two branches are also different in the sense that instruction #4 uses a relative address whereas instruction #6 uses an absolute address. This is perhaps better stated by examining their semantics: the unconditional branch is the easiest to consider, it performs the operation

$$PC \leftarrow (imm : 00_{(2)})$$

which in this specific case means

$$PC \leftarrow \texttt{done}$$

in that it simply sets the value of *PC* to the address `done`. Because of the subtlety of the ISA discussed in Chapter 5 which means that the least-significant two bits of the address are not stored, the assembler has to do some minor translation work. For example, assume that the program started at address 0 so that `done` is actually located at address 28. In this case, the assembler would encode the instruction `j done` (or `j 28`, since they are equivalent) as the 32-bit machine code output

$$00001000000000000000000000000111_{(2)}.$$

Notice that the most-significant six bits are $000010_{(2)}$ which mean the opcode implies a `j` or jump instruction. The other twenty six bits represent the absolute target of the jump. But the constant is actually $7_{(10)}$ rather than the expected $28_{(10)}$. This is because we know the processor will append zero bits for us at the least-significant end of the address in order to multiply it by four; this will produce the required address of $28_{(10)}$. Using a similar reasoning, the conditional branch performs the operation

$$\text{if } GPR[rs] \leq 0, PC \leftarrow PC + ext(off : 00_{(2)})$$

which in this specific case means

$$\text{if } GPR[4] \leq 0, PC \leftarrow PC + 8.$$

The actual offset used is maybe a bit confusing: why move the *PC* forward eight bytes (i.e., two instructions) ? Again recall the discussion in Chapter 5: when we perform a branch that uses an offset as the target, we need to keep in mind that the *PC* will have already been incremented during the fetch stage of the execution cycle. Thus, the actual offset is off-by-one compared to what one might expect: in this case, to get from the branch to the target we move the *PC* forward two more instructions, i.e., 8 bytes, instead of three instructions which would be more obvious. Because the branch uses an offset from the current *PC* value rather than an absolute address, it does not matter where the program is located in memory: the instruction encoding is the same in any case. The assembler would encode the in-

struction `blez $4, take` (or `blez $4, 8`, since they are equivalent) as the 32-bit machine code output

$$00011000100000000000000000000010_{(2)}.$$

Notice that the most-significant six bits are $000110_{(2)}$ which mean the opcode implies a `blez` or branch if less than or equal to zero instruction. The next five bits specify the register we test, in this case we want *GPR*[4] so the value is $00100_{(2)}$. The next five bits are zero per the encoding scheme, the final sixteen bits represent the branch offset. Notice that again the offset $8_{(10)}$ is encoded as $2_{(10)}$ because we know the processor will append zero bits for us at the least-significant end.

10.5.2.4 Special Instructions

The final class are **special instructions** which do not really fit anywhere else. Hopefully the `nop` instruction is self explanatory; it simply performs the null operation or no operation: it does not alter the processor state, execution simply skips onto the next instruction. The `syscall` and `eret` instructions enable the operation of an exception or interrupt mechanism. Both are discussed more in Chapter 12 and Appendix A.

10.5.3 Using Assembler Aliases

In comparison to the expressiveness granted by high-level languages such as C, the structure of assembly language and limited set of instructions available in the processor ISA sometimes make writing low-level programs somewhat cryptic. For example, in C one might easily write and understand the statement `a = 0` as assigning some variable `a` the value zero. In MIPS32 assembly language this might be written as the machine instruction `add $1, $0, $0`. Even with some context which is missing here, this instruction is harder to understand. We first need to know that `a` is stored in register *GRP*[1] (rather than referring to it by name); in order to understand the meaning of the instructing we also need to know that *GPR*[0] always stores zero and hence adding *GPR*[0] to itself and storing the result in *GPR*[1] is essentially setting `a` to zero.

In an effort to make life easier for the programmer, most assemblers allow what are commonly called instruction and register **aliases**, or in the specific cases of instructions, **pseudo-instructions**. We already looked roughly at pseudo-instructions in Chapter 5 when we discussed the fact that `nop` need not be implemented as a special case in the processor hardware because there are plenty of instructions whose semantics are the same. The example used there was an `sllv` instruction that targeted *GPR*[0] to throw away the result and hence performed no operation, i.e., a `nop`. In this case, the assembler would spot the `nop` in the assembly language pro-

gram but utilise the appropriate `sllv` instruction encoding. Any assembler is free to implement similar types of translation that could help the programmer. Consider for example the fact that the reduced MIPS32 ISA has no native instruction for bitwise NOT. We can obviously still perform a NOT operation using NOR since

$$\neg x \equiv x \barwedge x$$

but it would be clearer and more meaningful for the programmer to use a `not` mnemonic. Thus, we could introduce a `not` pseudo-instruction which, when encountered by the assembler, is translated into appropriate use of `nor`.

In much the same way, it is quite hard for a programmer to keep track of general-purpose register usage given that so far we have referred to such registers simply with a meaningless number. Table 10.2 details a list of register aliases common in MIPS32 assemblers. The description column is largely unimportant at the moment, we will describe some of the more specific register later. However, realise that instead of writing `$31` to refer to a return address, used for example by the `jal` instruction, we can write `$ra` which is then translated by the assembler into the correct register number.

Two common pseudo-instructions which we will utilise in the future, but which are not implemented directly by the processor or specified in the ISA, are `li` and `la`. Consider an example of their use:

```
.data

A:    .word 0
      .word 1
      ...

.text

main: la  $2, A      # set GPR[2] to &A
      li  $3, 4      # set GPR[3] to  4
      add $3, $2, $3 # set GPR[3] to &A+4
      lw  $4, 0($3)  # set GPR[4] to MEM[&A+4] = A[1]
```

The `la` pseudo-instruction loads the address of a label operand into the target register. Thus, after executing the first instruction the register *GPR*[2] holds the address of label `A`. In theory, we should be able to write something like `addi $2, $0, A` but since the address of `A` can be a full 32-bit number and the `addi` instruction can only accommodate a 16-bit immediate this is problematic. Instead, the `la` pseudo-instruction can act more intelligently by *constructing* the value in *GPR*[2], potentially expanding to several instructions that form the value in stages. A similar argument is true for the `li` pseudo-instruction which loads a register with an immediate value. After execution of the second instruction the register *GPR*[3] holds the value 4. Again, we might have written something like `addi $3, $0, 4` but if the immediate value is large we run into the same problem as described above; the use of a pseudo-instruction solves this problem while hiding the details from the programmer.

Register Number	Register Name	Description
0	0	Constant 0
1	*at*	Reserved
2	*v*0	Return value
3	*v*1	Return value
4	*a*0	Argument
5	*a*1	Argument
6	*a*2	Argument
7	*a*3	Argument
8	*t*0	Caller-save temporary
9	*t*1	Caller-save temporary
10	*t*2	Caller-save temporary
11	*t*3	Caller-save temporary
12	*t*4	Caller-save temporary
13	*t*5	Caller-save temporary
14	*t*6	Caller-save temporary
15	*t*7	Caller-save temporary
16	*s*0	Callee-save temporary
17	*s*1	Callee-save temporary
18	*s*2	Callee-save temporary
19	*s*3	Callee-save temporary
20	*s*4	Callee-save temporary
21	*s*5	Callee-save temporary
22	*s*6	Callee-save temporary
23	*s*7	Callee-save temporary
24	*t*8	Caller-save temporary
25	*t*9	Caller-save temporary
26	*k*0	Reserved
27	*k*1	Reserved
28	*gp*	Reserved
29	*sp*	Stack pointer (*SP*)
30	*fp*	Frame pointer (*FP*)
31	*ra*	Return address

Table 10.2 A table of MIPS32 register names, aliases and standard uses.

10.5.4 Using Assembler Directives

Statements which relate to machine instructions, either directly or via some alias, typically generate some output: they are a command by the programmer to translate the instruction into the machine code. In contrast, a **directive** is a command to the assembler itself rather than an instruction for it to translate.

The typical use for such commands is injection of something other than an instruction into the machine code. The most simple directive of this form has the mnemonic `.space` and accepts one operand called, for example, n. Using the directive

```
.space 8
```

instructs the assembler to dispense with the normal action of producing an encoded 32-bit machine instruction from the statement, and to instead produce n bytes of empty space or padding; the statement above inserts a total of eight padding bytes.

A similar directive with slightly different behaviour is `.align` which injects a different amount of padding depending on the address it is located at. Given an operand n, the goal is to align the next statement so its address is a multiple of 2^n bytes. So, for example, if we write

```
.align 2
```

we want the address of the next instruction to be a multiple of $2^2 = 4$ bytes. If the directive was encountered when the address counter was 5 for example, this would result in three padding bytes being inserted so that the next statement would be generated with the address counter at 8 (which is a multiple of 4). However, if the directive was encountered when the address counter was 6, only two padding bytes are required.

As well as padding, one can inject constant values into the generated machine code. For example, the `.half` and `.word` directives both take a list of values and injects them, as 16-bit half-words and 32-bit full-words respectively, instead of encoded instructions. For example, consider:

```
.half 10, 20, 30, 40
.word 10, 20, 30, 40
```

Both directives are self-aligning in that they automatically place their output on a multiple of a half-word or word respectively. Similar directives, with the mnemonics `.float` and `.double` exist for single and double precision floating point constants. To draw an analogy with the previous discussion of machine instruction encoding, for the first example above the assembler generates 64 bits of machine code output, i.e., four 16-bit values, that we might write as two 32-bit little-endian words

$$\begin{aligned}&0000000000001010000000000010100_{(2)}\\&0000000000011110000000000101000_{(2)}.\end{aligned}$$

In the second example, we get 128 bits or four 32-bit words of machine code output from the assembler which look like

$$\begin{aligned}&00000000000000000000000000001010_{(2)}\\&00000000000000000000000000010100_{(2)}\\&00000000000000000000000000011110_{(2)}\\&00000000000000000000000000101000_{(2)}.\end{aligned}$$

One can also inject text, in the form of strings of characters, in much the same way. The `.ascii` directive takes a string (which is again essentially a list of values) and injects characters as 8-bit bytes:

```
.ascii "constant"
```

Recall from Chapter 1 that ASCII gives a standard way to encode characters as binary strings. As such, we would expect the assembler to produce 64 bits of machine code for this string, written as two 32-bit little-endian words

$$01110011011011100110111101100011_{(2)}$$
$$01110100011011100110000101110100_{(2)}$$

where, for example, in the first word reading from the least-significant end the 8-bit chunks represent the characters 'c', 'o', 'n' and 's'. Using `.asciiz` instead of `.ascii` instructs the assembler to null-terminate the characters so that the result is compatible with the C model of how strings are represented. So given an n-byte string, `.asciiz` injects $n+1$ bytes where the last one is always zero.

You might be rightly thinking that injecting non-instructions into the machine code is dangerous: what if the processor were to try to execute them ? Clearly, since they are not encoded instructions it is difficult to say what the processor behaviour will be: basically it will decode the value as normal and act as if it were an instruction. This is unlikely to cause meaningful or desirable behaviour. Normally, the assembler simply leaves it to the programmer to ensure that control-flow can never reach non-instructions. However, it also offers a mechanism to segregate them; the `.text` and `.data` directive can be used to denote sections of instructions and non-instructions. This allows the assembler to perform some error checking, for example it might not normally allow a `.ascii` directive within the `.text` section, and to map the sections of instructions and non-instructions to suitable locations in the resulting machine code. In the case of simulation using SPIM, it has the added side benefit of allowing the simulator to print values differently for instruction and data sections so they are more meaningful to the programmer.

Finally, it is worth noting the significance of the `.globl` and `.extern` directives that we have so far used without explanation. The basic idea is that these directives allow us to control the visibility of labels. If we use a `.globl` directive followed by a label name, the symbol associated with that label will be marked as globally visible in the resulting object file; using `.extern` marks the symbol as external. This allows the linker to perform symbol resolution: if a symbol is marked as global in one object file and external in another, the linker can connect the two and resolve incomplete references to the external symbol into concrete references to the global symbol.

10.5.5 Peephole Optimisation

The term **peephole optimisation** in the context of assemblers usually means that we run a small window (or peephole) over the program and try to make localised performance improvements to instructions within the window. Because the window is small, usually just a few instructions, we are quite limited to what these improvements can be. We are also limited in the sense that by improving performance we cannot alter the meaning of the program: in particular, the assembler should not be responsible for bugs !

As an example of the sort of rewriting the assembler might perform, consider the instruction

```
...
add $2, $0, $2
...
```

which does nothing useful: it essentially adds zero to *GPR*[2]. While it might not be sensible for a human to write such an instruction, it is not uncommon for higher-level tools (or lazy humans) to generate them. The assembler can eliminate the instruction and improve performance as a result. As another example, consider the two instructions

```
...
addi $2, $2, 1
addi $2, $2, 1
...
```

Clearly there is no need for them to exist separately: incrementing *GPR*[2] and then incrementing it again can be achieved using a single instruction

```
...
addi $2, $2, 2
...
```

by rewriting one of the previous `addi` instructions and eliminating the other. Finally, consider branches of the form

```
    ...
    beq $2, $3, l0
l0: ...
```

There is no point in performing the equality test on *GPR*[2] and *GPR*[3] or the resulting branch since either way, control-flow drops through to the next instruction labelled `l0`. So we can eliminate the branch to give

```
    ...
l0: ...
```

Many other situations like the three presented above occur commonly in machine generated assembly language programs. By equipping the assembler with even basic rules for peephole optimisation, non-trivial improvements in performance are possible. This is particularly true if, for example, the improvements occur within a loop executed many times. These rules can often be merged with the set of rewriting rules used to realise the translation of pseudo-instructions into machine instructions.

```
1  .globl A
2  .globl n
3  .globl main
4
5  .data
6
7  A:    .space  64
8  n:    .word   16
9
10 .text
11
12 main: lw      $3, n($0)  # load n
13       add     $5, $0, $0 # set A[0] to 0
14       addi    $6, $0,  1 # set A[1] to 1
15       add     $7, $0, $0
16       addi    $8, $0,  4
17       sw      $5, A($7)  # store A[0]
18       sw      $6, A($8)  # store A[1]
19       addi    $4, $0,  2 # set i to 2
20 l0:   sub     $7, $3, $4
21       blez    $7, l1     # goto l1 if n <= i
22       add     $7, $5, $6 # set A[i] to A[i-1] + A[i-2]
23       addi    $8, $0,  4
24       mul     $8, $8, $4
25       sw      $7, A($8)  # store A[i]
26       add     $5, $0, $6 # set A[i-2] to A[i-1]
27       add     $6, $0, $7 # set A[i-1] to A[i  ]
28       addi    $4, $4,  1 # set i to i + 1
29       j       l0         # goto l0
30 l1:   addi    $2, $0, 10
31       syscall            # exit
```

Listing 10.10 MIPS32 implementation of a Fibonacci sequence generator.

10.5.6 Some Short Example Programs

Like most programming, learning to write assembly language is something which is best done by practise and example rather than simply reading about it. With this in mind, the following are some short example programs which are intended to demonstrate several common programming constructs used to solve real problems of roughly increasing complexity. The big caveat with these examples is that we have not yet looked at functions or function calls, deferring this subject until we discuss compilers in Chapter 11. Therefore, each example is presented as if it were the "whole program" even if in reality it should be a function within a program.

10.5.6.1 Fibonacci Numbers

The i-th number in the Fibonacci sequence is defined by

$$a_i = a_{i-1} + a_{i-2}$$

with $a_0 = 0$ and $a_1 = 1$. Thus, the first few numbers of the Fibonacci sequence are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. Writing a program to generate these numbers is fairly

simple: we essentially just need to write a loop which adds together the previous two numbers to generate the next number. Imagine that such a program should generate the first n Fibonacci numbers and store them in an array A, i.e., $A_i = a_i$ for values $0 \leq i < n$. An implementation of such a program is shown in Listing 10.10.

Firstly, notice that inside the data section we have used the `.space` directive to clear 64 bytes of memory for our uninitialised array; placing the label `A` before the directive allows us to refer to the array by name. We also allocate a single word, using `.word` and labelled `n`, to hold the input n, i.e., the number of Fibonacci numbers we should generate into `A`. Secondly, it is worth noting the register allocation used (roughly) throughout the program: we use register $GPR[3]$ or `$3` to hold the value of n; register $GPR[4]$ or `$4` to hold the value of i; registers $GPR[5]$ and $GPR[6]$ to hold the previous two Fibonacci numbers, i.e., A_{i-1} and A_{i-2}; and registers $GPR[7]$ and $GPR[8]$ or `$7` and `$8` to hold temporary or intermediate values.

We are now in a position to read and understand the program itself; in the text section, execution starts at the address labelled `main`. First we load the value of `n` and initialise the first two numbers of the Fibonacci sequence to 0 and 1. Storing A_0 and A_1 demands some care: note that we are storing 32-bit integers so the addresses of A_0 and A_1 are `&A` $+0$ and `&A` $+4$. We then initialise the loop counter i to 2, and enter the core loop. This loop works by first testing if the loop counter is larger than the limit n; we achieve this in a similar way to the test shown previously. That is, we subtract i from n and exit the loop, i.e., jump to the address labelled `l1`, if the result is less than or equal to zero. The body of the loop first adds the previous two Fibonacci together and stores the result A_i at address `&A` $+4i$. Notice that this would be easier if there was support for the scaled addressing mode discussed in Chapter 5 ! The program then shuffles the Fibonacci numbers around so that in the next loop iteration A_i and A_{i-1} become the two previous numbers. Finally, we increment the loop counter `i` and then jump back up to the address labelled `l0` for another iteration.

Of course, like any program which implements an algorithm, there are other options which we might have used instead. For example, we might prefer not to keep A_{i-1} and A_{i-2} in registers if they are at a premium but to load them from memory in each loop iteration; we might have preferred to keep track of the address `&A` $+4i$ by repeated addition rather than using a multiplication of i by four. However, the program works: we can execute it and on completion the array `A` contains the first `n` Fibonacci numbers.

10.5.6.2 Integer Square Roots

Probably you have met the square root function already in mathematics:

$$x = y^2 \longrightarrow y = \sqrt{x}$$

and probably you do not remember it as particularly friendly ! To make things easier for this example, we will consider only integer square roots: the problem is to find y

Input: An integer value x.
Output: The integer value $y = \lfloor \sqrt{x} \rfloor$.

```
1  SQRT(x)
2      y ← 0
3      z ← x + 1
4      while y + 1 ≠ z do
5          m ← ⌊(y + z)/2⌋
6          if m² ≤ x then
7              y ← m
8          else
9              z ← m
10     return y
11 end
```

Algorithm 10.2: An algorithm for computing integer square roots using binary search.

such that $y = \lfloor \sqrt{x} \rfloor$. In other words, find the largest integer y such that $y^2 \leq x$. Also assume that x is positive and fits into a general-purpose register, i.e., $0 \leq x < 2^{32}$, and that $0 \leq y^2 < 2^{32}$.

About the most basic method to use would be to simply use brute force search: loop through increasing values of y until y^2 exceeds x. This is far from the best approach to solving the problem you could take; for large x it will clearly be *very* inefficient ! Even though there are plenty of more involved methods to be more efficient, let's consider improving on brute force search by employing binary search instead. Algorithm 10.2 details how this might work; essentially we initialise the search limits and then search until we narrow in on the result.

We can adopt a now fairly familiar approach to implementing the algorithm; the result is shown in Listing 10.11. The data section defines 32-bit words labelled `x` and `y` to represent the input and output of the algorithm; in this case we initialise $x = 256$ so we expect the output to be $y = 16$. We use register *GPR*[3] or `$3` to hold the value of x; register *GPR*[4] or `$4` to hold the value of y; register *GPR*[5] or `$5` to hold the value of z; register *GPR*[6] or `$6` to hold the value of m; and registers *GPR*[7] and *GPR*[8] or `$7` and `$8` to hold temporary or intermediate values. Again, in the text section execution starts at the address labelled `main`. We start by loading the value of x and initialising the search limits before entering the loop. The loop structure is similar to the previous example in the sense that the first thing we do is test whether $y + 1 = z$: if it does, we branch to `l3` to terminate the loop, otherwise we execute the loop body. The loop body starts immediately by computing m (using a right-shift in place of the division by 2) and then performs a test to check whether to replace the upper or lower limit, and update the right one accordingly (this is similar to the example where we computed the maximum of two numbers). Finally,

```
1  .globl x
2  .globl y
3  .globl main
4
5  .data
6
7  x:    .word   256
8  y:    .word   0
9
10 .text
11
12 main: lw      $3, x($0)  # load x
13       addi    $4, $0,  0 # set y to 0
14       addi    $5, $3,  1 # set z to x+1
15 l0:   addi    $7, $4,  1
16       sub     $7, $7, $5
17       beqz    $7, l3     # goto l3 if y+1 == z
18       add     $6, $4, $5
19       addi    $7, $0,  1
20       srlv    $6, $6, $7 # set m to ( y + z ) / 2
21       mul     $7, $6, $6
22       sub     $7, $7, $3
23       blez    $7, l2     # goto l2 if m^2 <= x
24 l1:   add     $5, $6, $0 # set z to m
25       j       l0         # goto l0
26 l2:   add     $4, $6, $0 # set y to m
27       j       l0         # goto l0
28 l3:   sw      $4, y($0)  # store y
29       addi    $2, $0, 10
30       syscall            # exit
```

Listing 10.11 MIPS32 implementation integer square root using binary search.

we branch back to `l0` for another iteration. When the loop finally terminates, the program stores the result y and exits.

10.5.6.3 Bubble-Sort

As the last example we look at a basic sorting algorithm: the so-called bubble-sort in Algorithm 10.3 details a standard presentation of the algorithm which sorts an array a of n numbers into ascending order. Although things are starting to get a bit more complicated (there are two nested loops now for example), we can approach the implementation in exactly the same way as shown in Listing 10.12. We start with a data section which includes space for the array A, which is initialised to $A = (8,5,7,1,3,4,6,2)$, and the size n which in this case is 8. We reserve register $GPR[2]$ or `$2` to hold the exit flag f which tells us if the array is sorted prematurely; register $GPR[3]$ or `$3` holds n; registers $GPR[4]$ and $GPR[5]$ or `$4` and `$5` hold the loop counters i and j; registers $GPR[6\ldots10]$ or `$6` to `$10` hold temporary or intermediate values. Execution starts in the text section at the address labelled `main`. Things are now maybe a little verbose to explain line-per-line; hopefully the comments make the code fairly self-explanatory.

Input: An array A of length n.
Output: The sorted array A.

```
1  BUBBLE-SORT(A,n)
2      for i = n − 1 downto 0 do
3          f ← 0
4          for j = 0 upto i − 1 do
5              if A_{j+0} > A_{j+1} then
6                  t ← A_{j+0}
7                  A_{j+0} ← A_{j+1}
8                  A_{j+1} ← t
9                  f ← 1
10         if f = 0 then
11             return
12     return
13 end
```

Algorithm 10.3: An algorithm for sorting an array in-place using bubble-sort.

10.5.7 The Forward Referencing Problem

Consider the implementation of bubble-sort in Listing 10.12; we replicate this implementation below but focus only on the beginning and end of the code, cutting out the middle section:

```
main: lw      $3,  n($0)  # load n
      addi    $4,  $3, -1 # set i to n - 1
l0:   blez    $4,  l4     # goto l4 if i <= 0
      add     $2,  $0, $0 # set f to 0
      ...
      addi    $4,  $4, -1 # set i to i - 1
      j       l0          # goto l0
l4:   addi    $2,  $0, 10
      syscall             # exit
```

The first `blez` branch in this shortened fragment uses label `l4` as a target; the label is located later on in the program, toward the end. This is called a **forward reference**. The second `j` branch uses label `l0` as a target; the label is located earlier on in the program, toward the start. This is called a **backward reference**. Given the assembler will scan the program from start to finish, backward reference to a label causes no problem since at the point the label is used in an instruction, the assembler has already encountered the label definition and can therefore correctly recover the label address. However, a forward reference to a label is more problematic since at the point the label is used, the assembler does not yet know the label address: it has not encountered the label yet in the program. This is called the **forward referencing**

```
.globl A
.globl n
.globl main

.data

A:     .word   2
       .word   6
       .word   4
       .word   3
       .word   1
       .word   7
       .word   5
       .word   8

n:     .word   8

.text

main: lw      $3,  n($0)  # load n
      addi    $4,  $3, -1 # set i to n - 1
l0:   blez    $4,  l4     # goto l4 if i <= 0
      add     $2,  $0, $0 # set f to 0
      add     $5,  $0, $0 # set j to 0
l1:   sub     $6,  $4, $5
      blez    $6,  l3     # goto l3 if i <= j
      addi    $6,  $0,  4
      mul     $6,  $6, $5
      addi    $7,  $6,  4
      lw      $8,  A($6)  # load A[j  ]
      lw      $9,  A($7)  # load A[j+1]
      sub     $10, $8, $9
      blez    $10, l2     # goto l2 if a[j] <= a[j+1]
      sw      $9,  A($6)  # store A[j+1] (swap)
      sw      $8,  A($7)  # store A[j  ] (swap)
      addi    $2,  $0,  1 # set f to 1
l2:   addi    $5,  $5,  1 # set j to j + 1
      j       l1          # goto l1
l3:   blez    $2,  l4
      addi    $4,  $4, -1 # set i to i - 1
      j       l0          # goto l0
l4:   addi    $2,  $0, 10
      syscall             # exit
```

Listing 10.12 MIPS32 implementation of bubble-sort.

problem. So in the example above, a naive assembler cannot encode the `blez` branch since at that point in the program it has not encountered the `l4` label.

The usual way to solve this problem is to extend our naive assembly process, which we might call one-pass in the sense that it scans the program once from start to finish, into a two-pass version. During the first pass of our two-pass assembly process, the assembler scans the program and discards everything apart from label definitions. It uses these label definitions to build a **symbol table** in memory which holds each label name and the corresponding address. After finishing the first pass, the symbol table holds every label in the program. During the second pass, the assembler re-scans the program and this time performs the process of encoding statements; since it can now look-up label addresses in the symbol table created in

the first pass rather than relying on having encountered them in the second pass, the forward referencing problem is no longer a problem.

Of course, the fact that we now perform two passes over the program means that we need to perform two scans of the program i.e., parse the program twice. The positive of physically parsing the same program twice is that the two-pass assembler needs less memory (it need not hold the entire program in memory); the negative is that for large programs the repeated parsing effort can be problematic. As an alternative, some assemblers make the opposite trade-off by parsing the program once, holding it in memory in some intermediate form, and then applying the two passes without the need for re-parsing.

10.5.8 An Example Assembler

Assemblers are interesting from the point of view of computer architecture because, roughly speaking, they are the lowest level where human written programs meet the physical hardware: they commonly represent the programming interface to a given processor. They are also interesting because unlike more complex software, a basic but working assembler can be constructed in only a few hundred lines of C code. In this section we attempt to demonstrate this fact by presenting a skeleton assembler implementation (in the sense that some trivial parts are omitted) with the goal of having enough of the assembler in place to assemble the bubble-sort program in Listing 10.12.

10.5.8.1 Support Structures and the ISA Table

To support everything, Listing 10.13 details a number of common structures used in the assembler to represent labels (the `labl` structure), statements (the `stmt` structure) and instructions in the ISA (the `inst` structure). A `labl` holds the label name (or identifier) and the address at which it is defined; this structure is mainly used to represent entries in the symbol table. A `stmt` represents a parsed assembly language statement; it holds a count of each of the number of labels, mnemonics and operands parsed and their lexical representation (i.e., the raw strings that are parsed). An `inst` is a description of an instruction in the ISA; it holds the instruction mnemonic (as written in a program), opcode information (i.e., the `opcode` and `funct` fields), the number of operands and the type of those operands.

The `inst` structure is particularly important since it allows us to write what is often termed a **table driven assembler**. Such a design holds a single table which describes the ISA and drives the rest of the assembly process: everything else is generic so that adding or altering instructions means simply changing the table rather than the code. Listing 10.14 shows the table for our assembler. Each line initialises an `inst` structure for an instruction in the reduced MIPS32 ISA. Most of the table should be intuitive: the entry for `add` lists the mnemonic `add`; it uses a single 32-

```
typedef struct __labl
{
  struct __labl* next;
  struct __labl* prev;

            char  name[ MAX_CHAR ];
             int  addr;
}
labl;

typedef struct __smnt
{
  int  labl_n;
  char labl_s[ MAX_LABL ][ MAX_CHAR ];

  int  mnem_n;
  char mnem_s[ MAX_MNEM ][ MAX_CHAR ];

  int  oper_n;
  char oper_s[ MAX_OPER ][ MAX_CHAR ];
}
smnt;

typedef struct __inst
{
  char mnem_s[ MAX_CHAR ];

  int  code;

  int  oper_n;
  int  oper_t[ MAX_OPER ];
}
inst;
```

Listing 10.13 A C code fragment which describes support structures for the example assembler.

bit constant `OPER_ADD` to hold the encoded constants $000000_{(2)}$ and $100000_{(2)}$ which form the `opcode` and `funct` fields; it shows that the statement should have three operands; and most importantly it shows that the operands should all be registers. The operand type array is perhaps the most important. For `add` this array was simple, for other instructions things get more involved. For example, the entry for `addi` shows that the instruction takes three operands, the first two of which are registers and the third is a 16-bit immediate constant. The entry for `beq` shows that the instruction takes three operands, the first two of which are registers and the third is a 16-bit immediate offset which needs word aligning applied to it.

10.5.8.2 A Unified Parser Function

Before we worry about dealing with assembly language statements, we need to parse them somehow. Given that the assembler operates in a two-pass manner, we would rather not have two very similar functions for the parsing functionality. As such, Listing 10.15 shows a unified parsing function which accepts a file pointer and a

```
inst tab_inst[] = {{    "add", CODE_ADD,     3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {    "sub", CODE_SUB,     3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {    "mul", CODE_MUL,     3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {   "mult", CODE_MULT,    2, {         OPER_RS,    OPER_RT      }},
                   {    "and", CODE_AND,     3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {     "or", CODE_OR,      3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {    "xor", CODE_XOR,     3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {    "nor", CODE_NOR,     3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {   "sllv", CODE_SLLV,    3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {   "srlv", CODE_SRLV,    3, {OPER_RD, OPER_RS,    OPER_RT      }},
                   {   "addi", CODE_ADDI,    3, {OPER_RT, OPER_RS,    OPER_IMM16   }},

                   {   "mfhi", CODE_MFHI,    1, {OPER_RD                           }},
                   {   "mflo", CODE_MFLO,    1, {OPER_RD                           }},
                   {   "mfc0", CODE_MFC0,    2, {OPER_RD, OPER_RT                  }},
                   {   "mthi", CODE_MTHI,    1, {OPER_RS                           }},
                   {   "mtlo", CODE_MTLO,    1, {OPER_RS                           }},
                   {   "mtc0", CODE_MTC0,    2, {OPER_RD, OPER_RT                  }},
                   {     "lw", CODE_LW,      3, {OPER_RT, OPER_IMM16, OPER_RS      }},
                   {     "sw", CODE_SW,      3, {OPER_RT, OPER_IMM16, OPER_RS      }},

                   {    "beq", CODE_BEQ,     3, {OPER_RS, OPER_RT,    OPER_OFF16_WA}},
                   {    "bne", CODE_BNE,     3, {OPER_RS, OPER_RT,    OPER_OFF16_WA}},
                   {   "blez", CODE_BLEZ,    2, {OPER_RS,             OPER_OFF16_WA}},
                   {   "bgtz", CODE_BGTZ,    2, {OPER_RS,             OPER_OFF16_WA}},
                   {   "bltz", CODE_BLTZ,    2, {OPER_RS,             OPER_OFF16_WA}},
                   {   "bgez", CODE_BGEZ,    2, {OPER_RS,             OPER_OFF16_WA}},
                   {      "b", CODE_B,       1, {                     OPER_OFF26_WA}},
                   {      "j", CODE_J,       1, {                     OPER_IMM26_WA}},
                   {     "jr", CODE_JR,      1, {OPER_RS                           }},
                   {    "jal", CODE_JAL,     1, {                     OPER_IMM26_WA}},

                   {    "nop", CODE_NOP,     0, {                                   }},
                   {"syscall", CODE_SYSCALL, 0, {                                   }},
                   {  "break", CODE_BREAK,   0, {
}}};
```

Listing 10.14 A C code fragment which implements the driver table for the example assembler.

function pointer as input: it uses the file pointer as a source to read the program from and the function pointer as a handler which processes the statement.

The actual parsing process relies heavily on standard C library functions. The main `while` loop reads lines of input using `fgets` until they are exhausted. Each line is then stripped of extraneous characters by processing it in reverse; each occurrence of the hash character is replaced by end of line to truncate comments, commas and parentheses are replaced by space and tabs are eliminated to produce a string of raw tokens separated by space. So for example the line

```
lw $2, 0($3) # this is a comment
```

is manipulated into

```
lw $2  0 $3
```

The tokens are then extracted by the `strtok` function which operates in a loop to fill in a `stmt` structure which is subsequently passed to the handler function for processing.

```
void parse( FILE* f, void ( *process )( smnt* s ) )
{
  char l[ MAX_CHAR ];
  smnt s;

  while( fgets( l, MAX_CHAR, f ) != NULL )
  {
    char* p;
    char* q;

    for( int i = strlen( l ) - 1; i >= 0; i-- )
    {
      if( l[ i ] == '\n' || l[ i ] == '#' )
        l[ i ] = '\0';
      if( l[ i ] == '\t' || l[ i ] == ',' ||
          l[ i ] ==  '(' || l[ i ] == ')' )
        l[ i ] =  ' ';
    }

    s.labl_n = 0;
    s.mnem_n = 0;
    s.oper_n = 0;

    for( q = l; ( ( p = strtok( q, " " ) ) != NULL ); q = NULL )
    {
      if( s.mnem_n == 0 )
      {
        if( p[ strlen( p ) - 1 ] == ':' )
          strcpy( s.labl_s[ s.labl_n++ ], p );
        else
          strcpy( s.mnem_s[ s.mnem_n++ ], p );
      }
      else
      {
        strcpy( s.oper_s[ s.oper_n++ ], p );
      }
    }

    process( &s );
  }
}
```

Listing 10.15 A C function which parses assembly language programs line by line and calls a function to handle the resulting statement.

10.5.8.3 Functions for Passes One and Two

With the parser able to invoke an arbitrary handler function for each line of input, it simply remains to construct such functions which implement the right behaviour during the first and second passes of the assembly process. These functions are detailed in Listing 10.16 and Listing 10.17 respectively. Each code fragment implements three functions: a statement handler function which is called by the parser to process a statement (for example `pass_one_statement`); an instruction handler which deals specifically with machine instructions (for example `pass_one_instruction`); and a directive handler function which deals specifically with assembler directives (for example `pass_one_directive`).

```
void pass_one_directive( smnt* s )
{
  if( !strcmp( s->mnem_s[ 0 ], ".word" ) )
  {
    for( int i = 0; i < s->oper_n; i++ )
    {
      int n = atoi( s->oper_s[ i ] );

      addr += 4;
    }
  }
}

void pass_one_instruction( smnt* s )
{
  inst* t = get_inst( s->mnem_s[ 0 ] );

  if     ( t == NULL              )
    error( "unknown instruction" );
  else if( t->oper_n != s->oper_n )
    error( "wrong operand count" );
  else
  {
    for( int i = 0; i < t->oper_n; i++ )
    {
      switch( t->oper_t[ i ] )
      {
        case OPER_RD    :
        case OPER_RT    :
        case OPER_RS    :
          if( !is_reg( s->oper_s[ i ] ) )
            error( "wanted register operand" );
          break;
        case OPER_IMM16 :
        case OPER_OFF16 :
        case OPER_IMM26 :
        case OPER_OFF26 :
          if( !is_num( s->oper_s[ i ] ) )
            error( "wanted numeric operand"  );
          break;
      }
    }

    addr += 4;
  }
}

void pass_one_statement( smnt* s )
{
  if( s->labl_n != 0 )
  {
    if( get_labl( s->labl_s[ 0 ] ) != NULL )
      error( "redefined label" );
    else
      add_labl( s->labl_s[ 0 ], addr );
  }

  if( s->mnem_n != 0 )
  {
    if( is_directive( s->mnem_s[ 0 ] ) )
      pass_one_directive( s );
    else
      pass_one_instruction( s );
  }
}
```

Listing 10.16 C functions for the first pass of the example assembler.

```
1  void pass_two_directive( smnt* s )
2  {
3    if( !strcmp( s->mnem_s[ 0 ], ".word" ) )
4    {
5      for( int i = 0; i < s->oper_n; i++ )
6      {
7        int n = atoi( s->oper_s[ i ] );
8
9        output( addr, n );
10
11       addr += 4;
12     }
13   }
14 }
15
16 void pass_two_instruction( smnt* s )
17 {
18   inst* t = get_inst( s->mnem_s[ 0 ] );
19
20   uint32 enc = t->code;
21
22   for( int i = 0; i < t->oper_n; i++ )
23   {
24     switch( t->oper_t[ i ] )
25     {
26       case OPER_RD       :
27         enc |= ( ( dec_reg( s->oper_s[ i ] )            ) & 0x0000003F ) << 11;
28         break;
29       case OPER_RT       :
30         enc |= ( ( dec_reg( s->oper_s[ i ] )            ) & 0x0000003F ) << 16;
31         break;
32       case OPER_RS       :
33         enc |= ( ( dec_reg( s->oper_s[ i ] )            ) & 0x0000003F ) << 21;
34         break;
35       case OPER_IMM16    :
36         enc |= ( ( dec_num( s->oper_s[ i ] )            ) & 0x0000FFFF )      ;
37         break;
38       case OPER_OFF16    :
39         enc |= ( ( dec_num( s->oper_s[ i ] ) - addr - 4 ) & 0x0000FFFF )      ;
40         break;
41       case OPER_IMM26    :
42         enc |= ( ( dec_num( s->oper_s[ i ] )            ) & 0x03FFFFFF )      ;
43         break;
44       case OPER_OFF26    :
45         enc |= ( ( dec_num( s->oper_s[ i ] ) - addr - 4 ) & 0x03FFFFFF )      ;
46         break;
47       case OPER_IMM16_WA :
48         enc |= ( ( dec_num( s->oper_s[ i ] )            ) & 0x0000FFFF ) >>  2;
49         break;
50       case OPER_OFF16_WA :
51         enc |= ( ( dec_num( s->oper_s[ i ] ) - addr - 4 ) & 0x0000FFFF ) >>  2;
52         break;
53       case OPER_IMM26_WA :
54         enc |= ( ( dec_num( s->oper_s[ i ] )            ) & 0x03FFFFFF ) >>  2;
55         break;
56       case OPER_OFF26_WA :
57         enc |= ( ( dec_num( s->oper_s[ i ] ) - addr - 4 ) & 0x03FFFFFF ) >>  2;
58         break;
59     }
60   }
61
62   output( addr, enc );
63
64   addr += 4;
65 }
66
67 void pass_two_statement( smnt* s )
68 {
69   if( s->mnem_n != 0 )
70   {
71     if( is_directive( s->mnem_s[ 0 ] ) )
72       pass_two_directive( s );
73     else
74       pass_two_instruction( s );
75   }
76 }
```

Listing 10.17 C functions for the second pass of the example assembler.

During the first pass, the handler functions deal mainly with constructing the symbol table and performing error detection. For example, the statement handler first checks if the statement contains any labels. If it does, it then checks if the label has already been defined and issues an error if it has; if the label has not been defined, it adds it to the symbol table. The handler then checks if the statement contains any mnemonics. If it does, the handler determines if the mnemonic is a machine instruction or a directive and calls the specific handler for that case. The instruction handler for the first pass simply checks if the mnemonic is a valid instruction, if there are the right number of operands, and if the operands are of the correct type. Note that after this checking, the address counter is incremented. The directive handler performs a similar duty, incrementing the address counter for each operand in the list supplied to `.word` directives.

During the second pass, things get a bit more involved since we need to perform the actual instruction encoding; the main differences are therefore in the instruction and directive handlers. The directive handler is the simplest: for each operand to a `.word` directive it outputs that operand (as a 32-bit machine code word) and increments the address counter. The instruction handler needs to consider encoding of machine instructions which is a harder task. Roughly speaking, the function starts constructing the encoding by fetching opcode information from the ISA table for the instruction under inspection. It then loops through each operand and performs a different action depending on the type (which is also detailed in the ISA table). Each action first decodes the operand from the lexical representation into a numeric value. This involves for example translating the register number `$3` into the value 3, translating the immediate value `0xA` into the value 10 or retrieving a label address from the symbol table. Given the numeric value of the operand, the assembler can shift the value into the right place and then update the constructed encoding. Notice that there are some subtleties when dealing with branch instructions which use immediate and offset addresses which might also need to be further manipulated to accommodate word alignment. Once the encoding process is finished, the assembler outputs the encoded instruction (as a 32-bit machine code word) and increments the address counter.

With these handlers in place, the assembler can be executed by invoking the parser with one function pointer (to `pass_one_statement`) to operate the first pass and call it again with a different function pointer (to `pass_two_statement`) for the second pass. Obviously the address counter needs to be reset before each invocation and some initialisation of things like the symbol table need to be performed before either.

10.5.8.4 Some Example Output

To demonstrate the assembler in action, Table 10.3 details the line-by-line output when the bubble-sort program is offered as input. Each line in the table lists the address, the machine code word at that address and the assembly language statement which generated the machine code. Obviously this is making some simplifying as-

Address	Machine Code	Assembly Language
00000000	00000002	A: .word 2
00000004	00000006	.word 6
00000008	00000004	.word 4
0000000C	00000003	.word 3
00000010	00000001	.word 1
00000014	00000007	.word 7
00000018	00000005	.word 5
0000001C	00000008	.word 8
00000020	00000008	n: .word 8
00000024	8C030020	main: lw $3, n($0)
00000028	2064FFFF	addi $4, $3, -1
0000002C	18800013	l0: blez $4, l4
00000030	00001020	add $2, $0, $0
00000034	00002820	add $5, $0, $0
00000038	00853022	l1: sub $6, $4, $5
0000003C	18C0000C	blez $6, l3
00000040	20060004	addi $6, $0, 4
00000044	70C53002	mul $6, $6, $5
00000048	20C70004	addi $7, $6, 4
0000004C	8CC8000A	lw $8, a($6)
00000050	8CE9000A	lw $9, a($7)
00000054	01095022	sub $10, $8, $9
00000058	19400003	blez $10, l2
0000005C	ACC9000A	sw $9, a($6)
00000060	ACE8000A	sw $8, a($7)
00000064	20020001	addi $2, $0, 1
00000068	20A50001	l2: addi $5, $5, 1
0000006C	0800000E	j l1
00000070	18400002	l3: blez $2, l4
00000074	2084FFFF	addi $4, $4, -1
00000078	0800000B	j l0
0000007C	2002000A	l4: addi $2, $0, 10
00000080	0000000C	syscall

Table 10.3 Example machine code output of the skeleton two-pass MIPS32 assembler; note that the address and machine code columns are written as 32-bit values in base-16.

sumptions. Firstly, the assembler is directly generating machine code so we need to assume that the program is placed at address zero, otherwise assumptions about the address of, for example, the array `A` will not hold. Secondly, we do not care here about the fact that the data section is placed before the text section and might therefore be executed by a processor that starts execution at address zero; we assume there is some mechanism to start execution at the `main` entry point at address $24_{(16)} = 36_{(10)}$. Finally, the implementation above is not complete: we have assumed the support functions are filled in appropriately to get any output at all !

10.6 Further Reading

- A.W. Appel and M. Ginsburg.
  Modern Compiler Implementation in C.
  Cambridge University Press, 2004. ISBN: 0-521-60765-5.
- R.L. Britton.
  MIPS Assembly Language Programming.
  Prentice-Hall, 2003. ISBN: 0-131-42044-5.
- J.R. Levine.
  Linkers and Loaders.
  Morgan-Kaufmann, 2000. ISBN: 1-558-60496-0.
- S.S. Muchnick.
  Advanced Compiler Design and Implementation.
  Morgan-Kaufmann, 1997. ISBN: 1-558-60320-4.
- J. Waldron.
  Introduction to RISC Assembly Language.
  Addison-Wesley, 1998. ISBN: 0-201-39828-1.

10.7 Example Questions

42. Some RISC processors require the programmer or compiler to guarantee that a register is not read from for a given number of cycles after the content is loaded from memory; if the guarantee is not met, the program may execute incorrectly.

a. Explain why this approach might be used.
b. Give **one** advantage and **one** disadvantage of the approach.

43. The full MIPS32 instruction set includes `lb` and `sb`, instructions that load and store 8-bit bytes (rather than 32-bit words like the corresponding `lw` and `sw` instructions). Using these instructions, consider a function written in MIPS32 assembly language:

```
l0: lb   $t0, 0($a0)
    beqz $t0, l1
    addi $a0, $a0,   1
    j         l0
l1: add  $v0, $a1, $0
l2: lb   $t0, 0($a1)
    sb   $t0, 0($a0)
    addi $a1, $a1,   1
    addi $a0, $a0,   1
    beqz $t0, l3
    j         l2
l3: sub  $v0, $a1, $v0
    addi $v0, $v0,  -1
    jr   $ra
```

The function uses a pass-by-register calling convention: it takes two arguments which are placed in `$a0` and `$a1` by the caller, the return address is placed in `$ra`; the function produces one return value that is placed in `$v0` before it finishes execution.

a. Ignoring the issue of delay slots (i.e., assuming that the assembler managed this automatically), explain in English what the purpose of the function is and how it works.
b. The function works using `lb` and `sb` to load and store one 8-bit byte at a time. Given the purpose of the function, describe how it could potentially be optimised by using `lw` and `sw` instead and what problems this could entail.

Chapter 11
Compilers

Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic.

– G. Hopper

Abstract In the previous chapter we started by arguing that writing programs in machine code was tedious and error prone, and what we really needed was an assembler and linker to do some of the work for us. As programs get larger and more complex, one can start to use the same argument against assemblers and linkers: often they are too low-level, and higher-level programming tools are required. This has led to the invention of a variety of high-level languages, such as C, and **compilers** which translate programs written in those languages into low-level versions suitable for processing by assemblers and linkers. In this chapter the aim is to focus on the aspects of compiler tools which relate to the processors that eventually execute their output. That is, we shy away from issues of language design and focus on, for example, effective selection of machine instructions to implement a given program.

11.1 Introduction

Compilation of a program is an act of translation between a high-level language such as C, and a low-level language such as assembly language. A typical modern compiler is at least logically organised into three parts or phases as shown in Figure 11.1. In the first phase of processing, the **front-end** of the compiler is responsible for parsing the source program and translating it into some intermediate representation. This description is typically less complex and more canonical than the original program. The intermediate representation is passed to the second phase of processing whereby an optimisation framework analyses the program and optimises it via various transformations which are typically generic (i.e., processor independent). For example, it might eliminate common sub-expressions so identi-

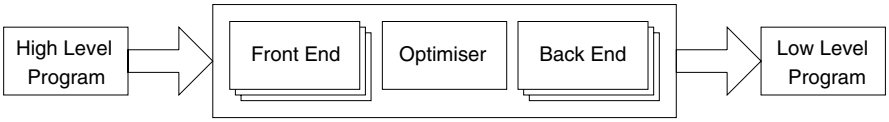


Figure 11.1 A rough block diagram of compiler components.

cal terms in an expression are only computed once; it might perform some form of strength reduction so that a computationally expensive operation (for example division) is replaced by a cheaper operation for specific cases (for example division by two becomes a right-shift); it might perform register allocation so that as many variables in the program as possible are held in the fastest parts of the memory hierarchy. The final phase of processing is performed by the **back-end** of the compiler. It is responsible for **code generation**, the translation of the program held in an intermediate representation into instructions for the target processor. This process is made harder by the issue of **instruction selection** (choosing the best combination of instructions to implement a given statement in the intermediate program) and **instruction scheduling** (placing the instructions in the best order to exploit features such as pipelining in the target processor).

It is interesting to note that until even quite recently, the output of compilers has been viewed as inferior to hand-written low-level programs. Partly, this is a case of bravado: the ability to write programs at a low level is often viewed as a mark of skill, and programmers enjoy the challenge of working at this level. Partly, it is based on fact however: for short programs written for simple processors, a good human programmer will usually be hard to beat. As software and processor complexity has increased, the use of high-level compilers has become increasingly common for all but the most performance critical applications. It is fair to attribute this change in attitude in large part to the development of the FORTRAN language and compiler at IBM by John Backus and others in the early 1950s. The FORTRAN language is used by the scientific computing community to this day. This fact is partly due to support for data-types such as complex numbers, but mainly because the FORTRAN compiler was probably the first which can be described as optimising the input program, particularly of loop constructs. Anecdotal evidence suggests that use of FORTRAN reduced program length by a factor of around twenty compared to assembly language alternatives of the time, while delivering largely equivalent performance. This trend was re-enforced by the development of the C programming language at Bell Labs by Dennis Ritchie in the early 1970s specifically for system-level programming and operating system development: this was a domain previously reserved for assembly language purists.

Unlike the subject of assemblers, compiler design and implementation is vastly too broad to go into detail here. The so-called “dragon book” by Aho et al. [1] is the standard reference in this area although Appel and Ginsburg [2] perhaps give a more practical introduction while Muchnick [47] focusses more deeply on vari-

ous optimisation techniques. To avoid glossing over the subject, we instead focus on elements of compilers which are most relevant to the platforms they target. As such, we consider the problem of bootstrapping a compiler for a particular platform, allocation of storage for variables in registers and within the stack and heap, basic code generation from high-level C statements to low-level assembly language, and the construction of a complete function call mechanism.

11.2 Compiler Bootstrapping and Re-Hosting

The result of decoupling the front-end from the back-end is, in an idealised sense, that we can write a front-end for each language we want to compile and a back-end for each target processor. By selecting one of each, we get a compiler for every language and every target processor using a common intermediate representation. Although maintaining the system in terms of compatibility between the phases is a challenge, this is a powerful feature that has in part enabled the GCC tool-chain, for example, to support a huge range of source languages and target processors.

In the context of compilers, the term **bootstrapping** refers to a "chicken or egg" type problem where we want to write a compiler in the same language it is intended to compile. The problem is, if we are to compile this implementation we need to exact thing we are trying to compile ! One approach is to implement the compiler entirely in assembly language. Historically, this was often the best option available. However, it means a lot of hard work. Another option is to bootstrap the compiler by writing an initial version in assembly language and then using this basic compiler to build successively complete compilers until one reaches the desired result:

1. Our starting point is an executable compiler C_0, originally written in language L_0, that executes on platform P_0. The compiler accepts programs written in language L_0 as input and produces executables that run on the same platform P_0 as output. At this point, C_0 can clearly compile its own source code S_0 (which was written in L_0). we call such a compiler **self-hosted**. A concrete example might be that L_0 is assembly language for some processor ISA such as a MIPS32.
2. We take the source code for S_0 for the compiler C_0 and change it to produce S_1. The change is such that the compiler S_1 describes can understand some slightly advanced version of language L_0 we call L_1; perhaps the programmer can write loop statements in L_1 for example. We can compile S_1 using C_0 to produce a new executable compiler C_1. At this point, C_1 is written in L_0, executes on platform P_0, accepts programs written in language L_1 as input and produces executables that run on the same platform P_0 as output.
3. We now take the source code for C_1 and re-write it in L_1 to produce S_2. Clearly this is easier than when it was originally written in L_0 since the new language contains some extra features we can take advantage of. Also, we can compile the resulting source code S_2 using C_1 to produce another executable compiler C_2. At this point, C_2 is written in L_1, executes on platform P_0, accepts programs written in language L_1 as input and produces executables that run on the same platform

P_0 as output. That is, the compiler C_2 is again self-hosted and obviously more powerful than our starting point C_0. To add further features to the compiler we simply iterate the process.

A related problem is that of re-hosting a compiler to a new target processor. In theory, this means simply changing the compiler back-end. However, this simply enables the compiler to generate code for that processor: we would like to execute the compiler on that processor as well:

1. Our starting point is an executable compiler C_0 for platform P_0 which accepts programs written in language L_0 as input and produces executables that run on the same platform P_0 as output. C_0 is termed a **native compiler** for P_0 since it produces executables for the same platform that it executes on itself. A concrete example might be an installation of `gcc` which you can run on a MIPS32-based processor and which compiles C programs to produce MIPS32 executables (that you can run on the same processor) as a result.
2. For the sake of argument, assume that the compiler is self-hosted, i.e., that the source code S_0 for compiler C_0 is written in language L_0. Thinking of C_0 as `gcc` means this step is no problem: it is written in C and can also compile C programs.
3. Next, we take the compiler source code S_0 and change it to produce a new set of source code S_1. The change is such that the back-end in S_0, which originally produced executables to run on platform P_0, is replaced with one that produces executables to run on platform P_1. We can compile S_1 with compiler C_0 on platform P_0. The resulting compiler C_1 executes on platform P_0 (we used C_0 to compile it), accepts programs written in language L_0 as input and produces executables that run on platform P_1 as output (we replaced the back-end in S_1 to ensure this). C_1 is termed a **cross compiler** since it produces executables for a different platform than it executes on itself.
4. Now we take the compiler source code S_1 and compile it with compiler C_1 again on platform P_0. The resulting compiler C_2 executes on platform P_1 (we used C_1 to compile it), accepts programs written in language L_0 as input and produces executables that run on platform P_1 as output (we replaced the back-end in S_1 to ensure this). In creating C_2 we now have a **native compiler** for P_1 since it again produces executables for the same platform that it executes on itself. We have **bootstrapped** the compiler for P_1 using the compiler for P_0.
   So, in short, we started with an installation of `gcc` for a MIPS32-based processor. Using only the source code and this initial compiler, we bootstrapped a `gcc` compiler for a totally different type of processor, an Intel Pentium for example.

11.3 Intermediate Representation

As touched on earlier, a compiler usually translates programs written in some high-level language into an intermediate representation that is more amenable to mechanical processing (rather than being read or written by a human). We usually term this

representation an **intermediate program** which consists of **intermediate instructions**. One way to view an intermediate program is as a sort of "symbolic" version of assembly language. For example, one might use 3-address style operations

$$x \leftarrow y \odot z$$

or memory access

$$x \leftarrow MEM[y]$$

or conditional branches

$$\text{if } x \odot y, \text{ goto } z$$

in a very similar way to how we were writing machine-level programs prior to the introduction of assembly language. However, there are a few differences between the two:

1. An intermediate program typically is not specific about where items of data are stored. Whereas in a machine-level program we need to talk about registers and memory addresses, a typical intermediate program views the data symbolically. For example, we might refer to a symbol called x in the intermediate program whereas in the machine-level program we would need to refer to $GPR[2]$ as storing x.
2. In a machine-level program, the operations we used really were machine instructions. An intermediate instruction, while more simple than a statement in the high-level language, does not necessarily map directly onto a machine instructions. For example, we might write an intermediate instruction $x \leftarrow y \ll 1$ to specify a left-shift of y by the constant 1. In the machine level program, we would have to have had a shift instruction that accepted a register operand and an immediate operand; in the intermediate program we can assume that some further translation will cope with this (e.g., by first loading 1 into a register and then performing a shift) if the only shift instruction takes two register operands.

Consider a simple example to illustrate these points: Algorithm 11.1 details a method for computing the Hamming weight of x. We can easily translate this algorithm into a high-level language program, for example a C program as shown in Listing 11.1. Notice that already we are being more concrete about operations as we move from higher-level to lower-level: for example where we write x_i in the algorithm, we need to be more specific and write the expression `( x >> i ) & 1` in the C program. Figure 11.2a shows the C program translated into an intermediate program; it still retains the symbolic names, but now the operations (where $\mathscr{I}$ is the instruction number) are simpler. For example, instead of writing the (composite) operation `( x >> i ) & 1` in one go, we split it into two simple intermediate instructions that use a temporary symbol t_0.

Within the intermediate program there are two features that demand definition. The first is the concept of a **basic block**. A basic block is a sequence of intermediate instructions where the first instruction in the sequence is the only entry point (i.e., the only one allowed to be the target of a branch from outside the block) and the

Input: An integer number x represented by n base-2 digits.
Output: The Hamming weight of x.

```
1 HAMMING-WEIGHT(x, n)
2     w ← 0
3     for i = 0 upto n − 1 step 1 do
4         if x_i = 1 then
5             w ← w + 1
6     return w
7 end
```

Algorithm 11.1: An algorithm to compute the Hamming weight of some value.

```
1  int HammingWeight( int x, n )
2  {
3    int w = 0;
4
5    for( int i = 0; i < n; i++ ) {
6      if( ( x >> i ) & 1 )
7        w = w + 1;
8    }
9
10   return w
11 }
```

Listing 11.1 The high-level Hamming weight algorithm converted to a C program.

$\mathscr{I}$	Intermediate Instruction
#0	$w \leftarrow 0$
#1	$i \leftarrow 0$
#2	l_0 : if $i \geq n$ goto l_2
#3	$t_0 \leftarrow x \gg i$
#4	$t_1 \leftarrow t_0 \wedge 1$
#5	if $t_1 \neq 1$ goto l_1
#6	$w \leftarrow w + 1$
#7	l_1 : $i \leftarrow i + 1$
#8	goto l_0
#9	l_2 : return w

(a) Intermediate instructions.

$\mathscr{B}$	$\mathscr{I}$	Intermediate Instruction
#0	#0	$w \leftarrow 0$
	#1	$i \leftarrow 0$
#1	#2	l_0 : if $i \geq n$ goto l_2
#2	#3	$t_0 \leftarrow x \gg i$
	#4	$t_1 \leftarrow t_0 \wedge 1$
	#5	if $t_1 \neq 1$ goto l_1
#3	#6	$w \leftarrow w + 1$
#4	#7	l_1 : $i \leftarrow i + 1$
	#8	goto l_0
#5	#9	l_2 : return w

(b) Intermediate basic blocks.

Figure 11.2 The Hamming weight C program converted to low-level intermediate representations.

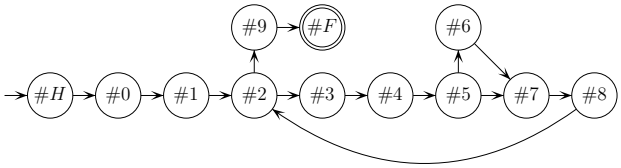


Figure 11.3 A Control-Flow Graph (CFG) relating to the intermediate representation of the Hamming weight program.

last instruction is the only exit point (i.e., the only one allowed to branch to outside the block). For all other instructions in the basic block, control-flow is straight-line. Figure 11.2b shows how the Hamming weight example can be sectioned into basic blocks where $\mathscr{B}$ is the basic block number. In each case, control-flow can only enter at the first instruction in the block and exit at the last instruction; control-flow is transferred either by an **explicit branch** (i.e., a branch instruction) or an **implicit branch** (i.e., when the processor would simply fetch the next instruction). The second feature is something we have briefly touched on already when talking about superscalar processors. A **trace** is a (non-cyclic) sequence of basic blocks that could be executed consecutively. Clearly whether the sequence *does* execute consecutively is data dependent, e.g., depends on conditional branches, and a single intermediate program might have many overlapping traces. In our example the traces are easy to extract as $[\#0,\#1,\#5]$, $[\#0,\#1,\#2,\#4]$ and $[\#0,\#1,\#2,\#3,\#4]$.

Finally, we can create some data structures that will help us analyse and transform the intermediate program. The central example of such a data structure is a **Control-Flow Graph (CFG)**, an example of which is detailed in Figure 11.3. A node in the graph represents an intermediate instruction, an edge in the graph represents control-flow; if there is an edge from node X to node Y, control-flow can pass from X to Y. The CFG could describe the control-flow over basic blocks to factor out the straight-line parts, but here we limit ourselves to intermediate instructions. We add an automatic header and footer node which represent the entry and exit points for the intermediate program.

11.4 Register Allocation

Every time our processor executes an instruction from a given program, it first needs to fetch some operands to operate on. With 3-address instructions we typically need to load two operands and store one result, although this might change depending on the ISA. If the operands are stored in memory, accessing them becomes a major performance bottleneck so ideally we keep as much of the working set as possible in the fastest areas of the memory hierarchy. Considering that the register file is the best place to store operands from a performance perspective, the problem is how to

$\mathscr{I}$	Intermediate Instruction	$succ[\mathscr{I}]$	$pred[\mathscr{I}]$	$use[\mathscr{I}]$	$def[\mathscr{I}]$
$\#H$		$\{\#0\}$	$\emptyset$	$\emptyset$	$\{x,n\}$
$\#0$	$w \leftarrow 0$	$\{\#1\}$	$\{\#H\}$	$\{0\}$	$\{w\}$
$\#1$	$i \leftarrow 0$	$\{\#2\}$	$\{\#0\}$	$\{0\}$	$\{i\}$
$\#2$	l_0 : if $i \geq n$ goto l_2	$\{\#3,\#9\}$	$\{\#1,\#8\}$	$\{i,n\}$	$\emptyset$
$\#3$	$t_0 \leftarrow x \gg i$	$\{\#4\}$	$\{\#2\}$	$\{x,i\}$	$\{t_0\}$
$\#4$	$t_1 \leftarrow t_0 \wedge 1$	$\{\#5\}$	$\{\#3\}$	$\{t_0,1\}$	$\{t_1\}$
$\#5$	if $t_1 \neq 1$ goto l_1	$\{\#6,\#7\}$	$\{\#4\}$	$\{t_1,1\}$	$\emptyset$
$\#6$	$w \leftarrow w+1$	$\{\#7\}$	$\{\#5\}$	$\{w,1\}$	$\{w\}$
$\#7$	l_1 : $i \leftarrow i+1$	$\{\#8\}$	$\{\#5,\#6\}$	$\{i,1\}$	$\{i\}$
$\#8$	goto l_0	$\{\#2\}$	$\{\#7\}$	$\emptyset$	$\emptyset$
$\#9$	l_2 : return w	$\{\#F\}$	$\{\#2\}$	$\{w\}$	$\emptyset$
$\#F$		$\emptyset$	$\{\#9\}$	$\emptyset$	$\emptyset$

Table 11.1 The first stage of analysis for the Hamming weight program: successor and predecessor sets, use and definition sets.

best allocate the (many) symbols in a program to the (fewer) registers in the register file.

Effective mechanisation of this process is vital for large programs since it is unrealistic for the programmer to perform the allocation by hand. Although there are other methods [53], one easy way to construct a solution relates to the problem of **graph colouring** [6, 13]. This technique dates back to early days of cartography: map makers wanted to draw maps where no neighbour countries had the same colour. If you imagine the countries are symbols in the program and the colours are register names, then essentially the two problems are the same. An overview of the approach is motivated by the following argument. Say we have a graph G and are trying to colour it using colours from the set C where $|C| = K$. Imagine G has some node X with $degree(X) < K$ then let $G\prime = G - \{X\}$, i.e the graph where we remove node X. If $G\prime$ is K-colourable, then so is G because when we add n then any neighbours have at most $K-1$ colours. That is, there is always a spare colour for X.

We will use the same common example throughout as in the previous section, i.e., Algorithm 11.1, which computes the Hamming weight of some value x, and from the CFG in Figure 11.3, we build two sets for each node as shown in Table 11.1. The **successor set** $succ[\mathscr{I}]$ is the set of nodes with edges from instruction $\mathscr{I}$, the **predecessor set** $pred[\mathscr{I}]$ is the set of nodes with edges to $\mathscr{I}$. For our example, this is fairly simple; we also include two further sets for each node: the **usage set** $use[\mathscr{I}]$ is the set of symbols used by $\mathscr{I}$, the **definition set** $def[\mathscr{I}]$ is the set of symbols defined by $\mathscr{I}$. Essentially these sets define what the data-flow, in contrast with control-flow, looks like. From the data-flow information we can compute liveness sets; that is, sets that tell us which symbols are **live-in** and **live-out**, or $in[\mathscr{I}]$

```
1  LIVENESS(G)
2      foreach n ∈ G do
3          in[n] ← ∅
4          out[n] ← ∅
5      repeat
6          foreach n ∈ G do
7              in′[n] ← in[n]
8              out′[n] ← out[n]
9              in[n] ← use[n] ∪ (out[n] − def[n])
10             out[n] ← ⋃_{s∈succ[n]} in[s]
11     until ∀n, in′[n] = in[n] and out′[n] = out[n]
12 end
```

Algorithm 11.2: Computation of live-in and live-out sets via iteration of data-flow equations.

$\mathscr{I}$	Intermediate Code	$in[\mathscr{I}]$	$out[\mathscr{I}]$
$\#H$		$\{0,1\}$	$\{x,n,0,1\}$
#0	$w \leftarrow 0$	$\{x,n,0,1\}$	$\{x,n,w,0,1\}$
#1	$i \leftarrow 0$	$\{x,n,w,0,1\}$	$\{x,n,w,i,1\}$
#2	l_0 : if $i \geq n$ goto l_2	$\{x,n,w,i,1\}$	$\{x,n,w,i,1\}$
#3	$t_0 \leftarrow x \gg i$	$\{x,n,w,i,1\}$	$\{x,n,w,i,t_0,1\}$
#4	$t_1 \leftarrow t_0 \wedge 1$	$\{x,n,w,i,t_0,1\}$	$\{x,n,w,i,t_1,1\}$
#5	if $t_1 \neq 1$ goto l_1	$\{x,n,w,i,t_1,1\}$	$\{x,n,w,i,1\}$
#6	$w \leftarrow w+1$	$\{x,n,w,i,1\}$	$\{x,n,w,i,1\}$
#7	l_1 : $i \leftarrow i+1$	$\{x,n,w,i,1\}$	$\{x,n,w,i,1\}$
#8	goto l_0	$\{x,n,w,i,1\}$	$\{x,n,w,i,1\}$
#9	l_2 : return w	$\{w\}$	$\emptyset$
$\#F$		$\emptyset$	$\emptyset$

Table 11.2 The second stage of analysis for the Hamming weight program: live-in and live-out sets.

and $out[\mathscr{I}]$, for each instruction $\mathscr{I}$. Intuitively, a symbol is live at a particular point in the program if it contains a useful value, i.e., one we cannot overwrite without changing the program meaning. There are several strategies for computing the liveness sets; the most elegant is via iteration of data-flow equations, which is detailed in Algorithm 11.2. For our example, we reach a solution after only a few iterations; the end result is shown in Table 11.2. Notice that constants 0 and 1 are live-in to the function; if these were variables rather than constants, this would indicate a case of use before definition, i.e., an error in the program. Maybe the only other feature of

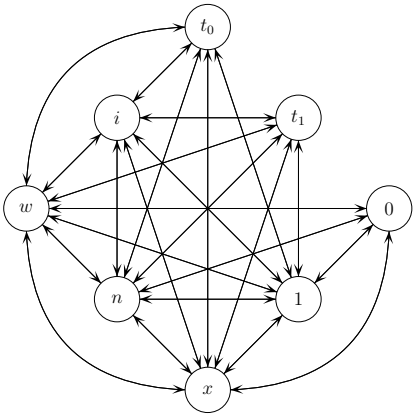


Figure 11.4 The interference graph for the Hamming weight program.

```
1  ALLOC-PREPARE(G,S,C)
2      G′ ← G
3      while |G|′ ≠ 0 do
4          t ←⊥
5          foreach n ∈ G′ do
6              if degree(n) < |C| then
7                  t ← n
8          if t =⊥ then
9              t ← SELECT-SPILL(G′)
10         G′ ← G′ − {t}
11         S ← push(S,t)
12 end
```

Algorithm 11.3: Using the interference graph to prepare the colouring stack.

note is that, for example, there is no edge between t_0 and t_1: their live ranges do not overlap.

Finally, we draw an **interference graph** where nodes represent symbols in the program and edges represent interference, i.e., the fact that two symbols cannot occupy the same register. We use the liveness information to do this: if symbols X and Y are live at the same time, they cannot be allocated the same register so edges (X,Y) and (Y,X) exist in the graph. We can draw the interference graph for our example as in Figure 11.4. To colour the graph (i.e., perform register allocation), the first step is to create a **colouring stack**. Algorithm 11.3 details the procedure; roughly speaking we repeatedly remove some node t with $degree(t) < |C|$ which

```
1  ALLOC-PREPARE(G,S,R)
2      while |S| ≠ 0 do
3          t ← pop(S)
4          C′ ← C
5          foreach n where (t,n) ∈ G do
6              C′ ← C′ − {colour(n)}
7          if |C|′ ≠ 0 then
8              colour(t) ←_R C′
9          else
10             PERFORM-SPILL()
11 end
```

Algorithm 11.4: Colouring the interference graph using the prepared colouring stack.

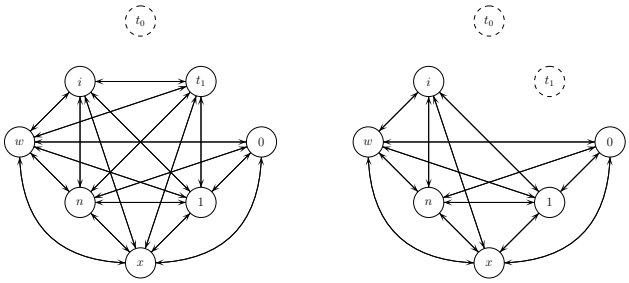
guarantees they are colourable. If we cannot find such a t, we carry on optimistically but note that this symbol might require spilling, a concept we examine later.

11.4.1 An Example Allocation

Consider an example where we try to colour our graph using a sub-set of general purpose registers $C = GPR[2 \ldots 7]$, i.e., $|C| = 6$. We start by initialising the colouring stack S to empty and starting with the original interference graph in Figure 11.4, iterate our algorithm:

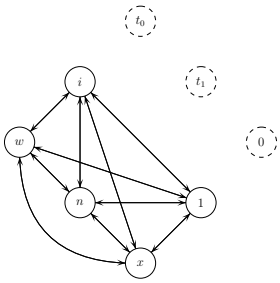
- Figure 11.5a removes t_0 so that $S = [t_0]$.
- Figure 11.5b removes t_1 so that $S = [t_0, t_1]$.
- Figure 11.5c removes 0 so that $S = [t_0, t_1, 0]$.
- Figure 11.5d removes 1 so that $S = [t_0, t_1, 0, 1]$.
- Figure 11.5e removes x so that $S = [t_0, t_1, 0, 1, x]$.
- Figure 11.5f removes n so that $S = [t_0, t_1, 0, 1, x, n]$.
- Figure 11.5g removes w so that $S = [t_0, t_1, 0, 1, x, n, w]$.
- Figure 11.5h removes i so that $S = [t_0, t_1, 0, 1, x, n, w, i]$.

Now that we have the colouring stack prepared, we can preform the colouring operation detailed in Algorithm 11.4; we pop nodes from the stack and try to colour them. In the case where we were optimistic in preparing the stack, we could find there is not a colour to assign; in this case we need to take some remedial action (i.e., spill a symbol) which we will describe later. Otherwise, we select a colour from the list. The colouring stack starts as $S = [t_0, t_1, 0, 1, x, n, w, i]$ and all symbols have no colour (or the null colour) assigned to them; then we proceed as follows:

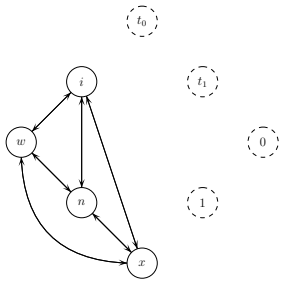


(a) Remove t_0.

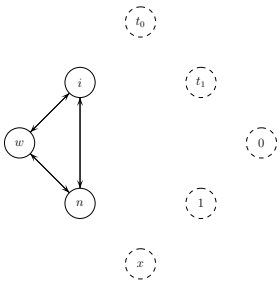
(b) Remove t_1.



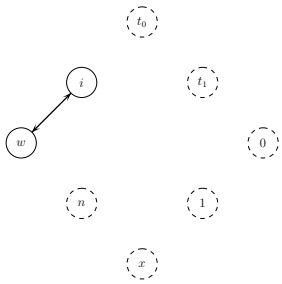
(c) Remove 0.



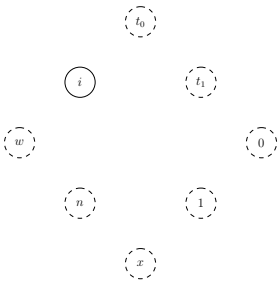
(d) Remove 1.



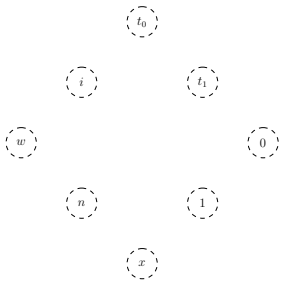
(e) Remove x.



(f) Remove n.



(g) Remove w.



(h) Remove i.

Figure 11.5 Preparation of the colouring stack for the Hamming weight program.

- Pop i, so that $S = [t_0, t_1, 0, 1, x, n, w]$, and assign it the colour $GPR[2]$ because none of x, n, w, t_0, t_1 and 1 remove any colours from $C\prime$.
- Pop w, so that $S = [t_0, t_1, 0, 1, x, n]$, and assign it the colour $GPR[3]$ because i removes $GPR[2]$ from $C\prime$.
- Pop n, so that $S = [t_0, t_1, 0, 1, x]$, and assign it the colour $GPR[4]$ because i and w remove $GPR[2]$ and $GPR[3]$ from $C\prime$.
- Pop x, so that $S = [t_0, t_1, 0, 1]$, and assign it the colour $GPR[5]$ because i, w and n remove $GPR[2]$, $GPR[3]$ and $GPR[4]$ from $C\prime$.
- Pop 1, so that $S = [t_0, t_1, 0]$, and assign it the colour $GPR[6]$ because i, w, n, and x remove $GPR[2]$, $GPR[3]$, $GPR[4]$ and $GPR[5]$ from $C\prime$.
- Pop 0, so that $S = [t_0, t_1]$, and assign it the colour $GPR[2]$ because w, n, and x remove $GPR[3]$, $GPR[4]$ and $GPR[5]$ from $C\prime$ but i and 0 do not interfere so $GPR[2]$ is still selectable.
- Pop t_1, so that $S = [t_0]$, and assign it the colour $GPR[7]$ because i, w, n, x and 1 remove $GPR[2]$, $GPR[3]$, $GPR[4]$, $GPR[5]$ and $GPR[6]$ from $C\prime$.
- Pop t_0, so that $S = []$, and assign it the colour $GPR[7]$ because i, w, n, x and 1 remove the colours $GPR[2]$, $GPR[3]$, $GPR[4]$, $GPR[5]$ and $GPR[6]$ from $C\prime$ but t_0 and t_1 do not interfere so $GPR[7]$ is still selectable.

After completion, we find that our colouring has produced the valid register allocation

$$\begin{aligned} colour(x) &= GPR[5] \\ colour(n) &= GPR[4] \\ colour(w) &= GPR[3] \\ colour(i) &= GPR[2] \\ colour(t_0) &= GPR[7] \\ colour(t_1) &= GPR[7] \\ colour(0) &= GPR[2] \\ colour(1) &= GPR[6] \end{aligned}$$

and we are done.

11.4.2 Uses for Pre-colouring

Use of graph colouring yields a natural way to deal with some important special cases for register allocation:

1. In some ISAs, the $2n$-bit result of $(n \times n)$-bit multiplication is stored in prescribed registers since a 3-address instruction cannot specify two targets.
2. Often, the function calling convention means incoming arguments and return values are passed in registers.

To solve these problems, we essentially just pre-colour some of the symbols. For example, say symbol X has to be coloured with register Y because of some constraint.

$\mathscr{I}$	Intermediate Instruction	$in[\mathscr{I}]$	$out[\mathscr{I}]$
$\#H$		$\{0,1\}$	$\{x,n,0,1\}$
#0	$w \leftarrow 0$	$\{x,n,0,1\}$	$\{x,n,w,0,1\}$
$\#S1$	$MEM[SP] \leftarrow w$	$\{x,n,w,0,1\}$	$\{x,n,0,1\}$
#1	$i \leftarrow 0$	$\{x,n,i,0,1\}$	$\{x,n,i,1\}$
#2	l_0 : if $i \geq n$ goto l_2	$\{x,n,i,1\}$	$\{x,n,i,1\}$
#3	$t_0 \leftarrow x \gg i$	$\{x,n,i,1\}$	$\{x,n,i,t_0,1\}$
#4	$t_1 \leftarrow t_0 \wedge 1$	$\{x,n,i,t_0,1\}$	$\{x,n,i,t_1,1\}$
#5	if $t_1 \neq 1$ goto l_1	$\{x,n,i,t_1,1\}$	$\{x,n,i,1\}$
$\#S2$	$w \leftarrow MEM[SP]$	$\{x,n,i,1\}$	$\{x,n,w,i,1\}$
#6	$w \leftarrow w+1$	$\{x,n,w,i,1\}$	$\{x,n,w,i,1\}$
$\#S3$	$MEM[SP] \leftarrow w$	$\{x,n,w,i,1\}$	$\{x,n,i,1\}$
#7	l_1 : $i \leftarrow i+1$	$\{x,n,i,1\}$	$\{x,n,i,1\}$
#8	goto l_0	$\{x,n,i,1\}$	$\{x,n,i,1\}$
$\#S4$	$w \leftarrow MEM[SP]$	$\{\}$	$\{w\}$
#9	l_2 : return w	$\{w\}$	$\emptyset$
$\#F$		$\emptyset$	$\emptyset$

Table 11.3 The second stage of analysis for the re-written Hamming weight program: live-in and live-out sets.

Before executing ALLOC-COLOUR we simply set $colour(X) = Y$ so that during the colouring process, other symbols cannot use Y.

11.4.3 Avoiding Error Cases via Spilling

When a large number of registers are in use, i.e., contain live symbols, we say that the level of **register pressure** is high; when we run out of registers some remedial action needs to be taken. If we could alter the processor, we could make the register file larger but this has some drawbacks. In particular it might ease register pressure but means increased area and requires a larger proportion of the instruction encoding to address the larger register file. More realistically we temporarily **spill** some register into main memory, for example onto the stack.

If we want to spill symbol X, we transform or re-write the program using two rules: wherever X is defined we store the value in memory, wherever X is used we load the value from memory again. This process alters the liveness analysis: the **live range** of X will be shorter and hence X will interfere with less other symbols. The question is, given a program *which* symbol should we spill ? This is normally guided by a rule of thumb called a **spill heuristic**. For example, we might

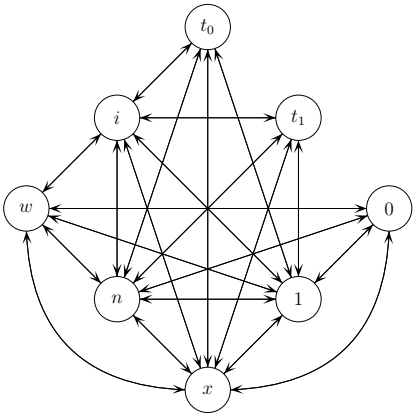


Figure 11.6 The interference graph for the re-written Hamming weight program.

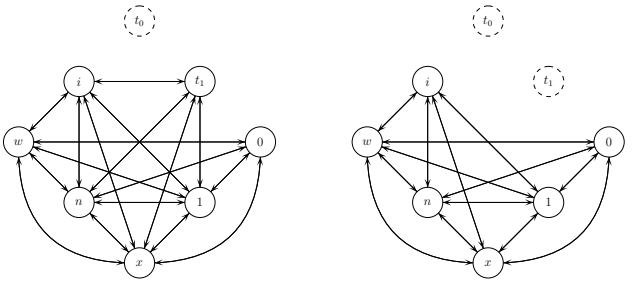
1. spill constants since re-loading them is cheap, or
2. spill symbols with long live ranges since this maximises the change in liveness analysis, or
3. spill symbols which are used the least since this minimises the cost of the extra memory access.

In the example program, the symbol w might be a reasonable selection: it has a long live range and is used somewhat infrequently. So we might elect to re-write the program so that w is spilt according to our rules; the end result is detailed in Table 11.3. The resulting interference is shown in Figure 11.6; notice that w no longer interferes with either t_0 or t_1 any more. As before, we can use an example to show the difference this has made: imagine we try to colour our new graph using a smaller sub-set of general-purpose registers $C = GPR[2 \ldots 6]$, i.e., $|C| = 5$. We start by initialising the colouring stack S to empty and iterate our algorithm:

- Figure 11.7a removes t_0 so that $S = [t_0]$.
- Figure 11.7b removes t_1 so that $S = [t_0, t_1]$.
- Figure 11.7c removes 0 so that $S = [t_0, t_1, 0]$.
- Figure 11.7d removes 1 so that $S = [t_0, t_1, 0, 1]$.
- Figure 11.7e removes x so that $S = [t_0, t_1, 0, 1, x]$.
- Figure 11.7f removes n so that $S = [t_0, t_1, 0, 1, x, n]$.
- Figure 11.7g removes w so that $S = [t_0, t_1, 0, 1, x, n, w]$.
- Figure 11.7h removes i so that $S = [t_0, t_1, 0, 1, x, n, w, i]$.

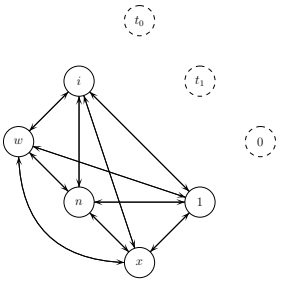
Now that we have the colouring stack prepared, we can again preform the actual colouring:

- The colouring stack starts as $S = [t_0, t_1, 0, 1, x, n, w, i]$ and all symbols have no colour (or the null colour) assigned to them.

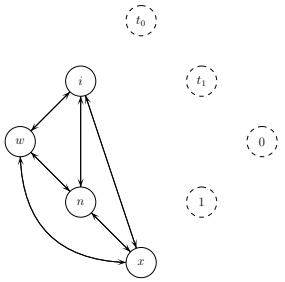


(a) Remove t_0.

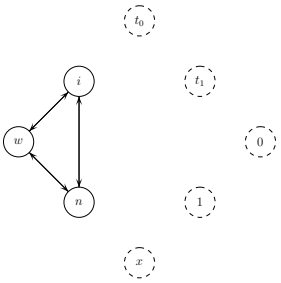
(b) Remove t_1.



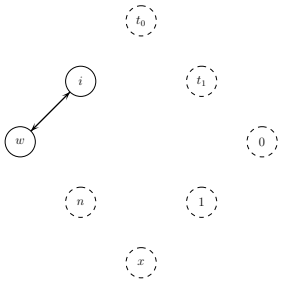
(c) Remove 0.



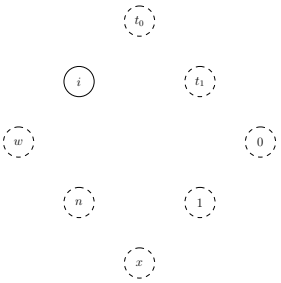
(d) Remove 1.



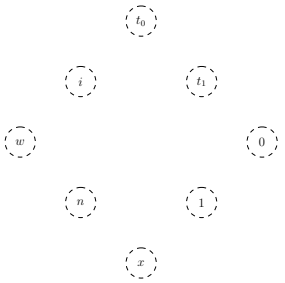
(e) Remove x.



(f) Remove n.



(g) Remove w.



(h) Remove i.

Figure 11.7 Preparation of the colouring stack for the rewritten Hamming weight program.

- Pop i, so that $S = [t_0, t_1, 0, 1, x, n, w]$, and assign it the colour $GPR[2]$ because none of x, n, w, t_0, t_1 and 1 remove any colours from $C\prime$.
- Pop w, so that $S = [t_0, t_1, 0, 1, x, n]$, and assign it the colour $GPR[3]$ because i removes $GPR[2]$ from $C\prime$.
- Pop n, so that $S = [t_0, t_1, 0, 1, x]$, and assign it the colour $GPR[4]$ because i and w remove $GPR[2]$ and $GPR[3]$ from $C\prime$.
- Pop x, so that $S = [t_0, t_1, 0, 1]$, and assign it the colour $GPR[5]$ because i, w and n remove $GPR[2]$, $GPR[3]$ and $GPR[4]$ from $C\prime$.
- Pop 1, so that $S = [t_0, t_1, 0]$, and assign it the colour $GPR[6]$ because i, w, n, and x remove $GPR[2]$, $GPR[3]$, $GPR[4]$ and $GPR[5]$ from $C\prime$.
- Pop 0, so that $S = [t_0, t_1]$, and assign it the colour $GPR[2]$ because w, n, and x remove $GPR[3]$, $GPR[4]$ and $GPR[5]$ from $C\prime$ but i and 0 do not interfere so $GPR[2]$ is still selectable.
- Pop t_1, so that $S = [t_0]$, and assign it the colour $GPR[3]$ because i, n, x and 1 remove $GPR[2]$, $GPR[4]$, $GPR[5]$ and $GPR[6]$ from $C\prime$ but t_1 and w do not interfere so $GPR[3]$ is a still selectable.
- Pop t_0, so that $S = []$, and assign it the colour $GPR[3]$ because i, n, x and 1 remove the colours $GPR[2]$, $GPR[4]$, $GPR[5]$ and $GPR[6]$ from $C\prime$ but t_0 and w don't interfere so $GPR[3]$ is still selectable.

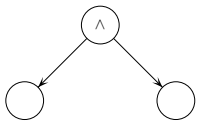
The difference this time is that we were able to allocate the same colour to w, t_0 and t_1 because after re-writing the program they do not interfere with each other, whereas before re-writing they did. After completion, we find that our colouring has produced the valid register allocation

$$\begin{aligned} colour(x) &= GPR[5] \\ colour(n) &= GPR[4] \\ colour(w) &= GPR[3] \\ colour(i) &= GPR[2] \\ colour(t_0) &= GPR[3] \\ colour(t_1) &= GPR[3] \\ colour(0) &= GPR[2] \\ colour(1) &= GPR[6] \end{aligned}$$

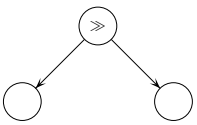
which means we have succeeded in performing register allocation using less registers than before as a direct result of spilling w.

11.5 Instruction Selection and Scheduling

Once the compiler has allocated symbols within the intermediate program to appropriate storage locations it can then translate the program, written in what we can view as a high-level source language, into an equivalent in a lower-level target language. In reality, there is probably a need to overlap these two tasks since decisions



Tile for
(a) $GPR[X] \leftarrow GPR[Y] \wedge GPR[Z]$.



Tile for
(b) $GPR[X] \leftarrow GPR[Y] \gg GPR[Z]$.

Figure 11.8 Two tiles representing 2-input, 1-output MIPS32 machine instructions. Note that the register addresses X, Y and Z are part of the template in the sense that they are filled in when the tiles are placed over the program tree.

made during register allocation sometimes have implications for translation and vice versa. Either way, we can view translation roughly as two processes:

1. **instruction selection**, which selects a suitable machine instruction to implement each intermediate instruction (or group thereof) in the intermediate program, and
2. **instruction scheduling**, which re-orders the instructions to optimise performance; one can view this as arranging the instructions we select in the "best" order.

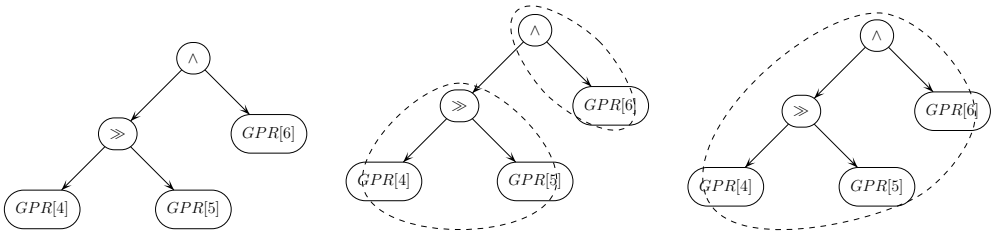
11.5.1 Instruction Selection

The instruction selection process translates intermediate operations into actual machine instructions; the tricky part to this is that is not always a one-to-one mapping between the two. That is, there is not always a single machine instruction that will implement a given intermediate instruction and, more annoyingly, there is sometimes *more* than one machine instruction (or sequence of) to choose from. A fairly standard way to approach the problem of instruction selection is to apply some form of **tiling** algorithm.

The basic idea is that we look at the intermediate program as a tree that describes the computation performed, and produce a tile for each machine instruction in the ISA. Each tile is sort of a small sub-tree and acts like a template for what the instruction can do. For example, the MIPS32 ISA includes two machine instructions

$$GPR[rd] \leftarrow GPR[rt] \gg GPR[rs]$$
$$GPR[rd] \leftarrow GPR[rs] \wedge GPR[rt]$$

From these descriptions we might generate two (fairly trivial) tiles as shown in Figure 11.8. The tiles are essentially small trees where the "blank" nodes show where we can combine tiles, for example to show that one tile generates a result that is used by another. The goal is to perform this combination in such a way that we cover the program tree with non-overlapping tiles. Given each tile relates to a machine in-



(a) Untiled program tree. (b) Tiling for MIPS32. (c) Tiling for ARM.

Figure 11.9 The original program tree and two different instruction selections resulting from the difference between two ISAs.

struction, tiling the program tree effectively acts to select machine instructions for the program. However, a given program tree might allow many potential tilings: we need a way to select the "best" one. To assist in this process we assign a weight to each tile. For example, the number of nodes in the tile might be a good weight, or perhaps an approximation of the machine instruction cost. An optimum tiling is such that the weights sum to the lowest possible value; an optimal tiling is where two adjacent tiles cannot be merged into a tile of lower weight.

Consider the computation of t_1 in the example Hamming weight program from Figure 11.2. For the sake of simplicity, if we assume that the symbols t_0, t_1, x, y, and i are allocated the registers $GPR[2]$ $GPR[3]$, $GPR[4]$, $GPR[5]$ and $GPR[6]$, then we can view computation of t_1 as a small tree as shown in Figure 11.9a: the leaf nodes are the symbols x, i and 1, the interior nodes are the operations that perform $t_0 \leftarrow x \gg i$ and $t_1 \leftarrow t_0 \wedge 1$; the root of the tree can be seen as generating t_1. As a result, we can tile our program tree by combining the tiles as shown in Figure 11.9b. As an aside, even if the symbol 1 is not in a register, MIPS32 has an instruction

$$GPR[rd] \leftarrow GPR[rs] \wedge imm$$

which helps us out, i.e., avoids the need for a third (pseudo) tile to move the value 1 into a register. However, consider what might happen if we had a significantly different ISA. For example the ARM process might includes an instruction with the semantics

$$GPR[rd] \leftarrow GPR[rn] \wedge (GPR[rm] \gg GPR[rs])$$

which is, roughly speaking, to provide pre-scaled operands that are useful as an addressing mode. This means on an ARM, we would expect to be able to tile the program tree with one tile as shown in Figure 11.9c.

Since for larger and more complex programs the tree will be (much) larger and more complex, we need a way to automate the process. A simple but surprisingly

effective strategy is maximal munch: we simply process the program tree in a top-down manner, covering as much of it as possible at each step:

1. Take the root node R of a tree and find the largest tile (i.e., the one with most nodes) T which can cover R.
2. Cover R (and possibly other nodes) with T; this creates a number of sub-trees S_i.
3. Recursively apply the algorithm from step 1 for each S_i.

Note that the algorithm generates instructions in reverse order, i.e., the instructions at the root should really go first ! Also note that we need to be careful about the ordering. For example, we probably need to preserve the order of accesses to memory so should take this into account when walking the tiled tree to actually emit the machine instructions.

11.5.2 Instruction Scheduling

The instruction scheduling process orders the machine instructions that we select; the aim is to select the "best" order, for example an order that improves the utilisation of PEs in a superscalar processor. The problem is, not all orderings of the machine instructions are valid: some will break the program due to instruction-level dependency ! Given we have already done the analysis on the program to section it into basic blocks, we can either work in a global way (i.e., re-order the blocks themselves) or a local way (i.e., re-order instructions within a block).

11.5.3 Scheduling Basic Blocks

The idea of **trace scheduling** is to order the basic blocks in a way that improves performance. Basically, we look for a trace (or set of traces) that we guess will be executed more often than the others: we guess that this trace will dominate performance so treat it with a higher priority with respect to optimisation. To do this we order the basic blocks in the trace we selected so they form straight-line code; if there is a branch from block X to block Y, we place block Y immediately after block X. This sometimes means the branches between the basic blocks can be eliminated. The strategy effectively optimises for the average case: as long as the trace we select is executed more often, performance is improved overall. However, in the (hopefully fewer) cases where our guess does not hold and another trace executes, we pay some penalty.

To guess which trace(s) are good candidates for ordering in sequence, we effectively need to predict which way branches will go; doing this at compile-time is **static branch prediction**, we are doing at compile-time what a processor that implements speculative execution might do at run-time. Since conditional branch

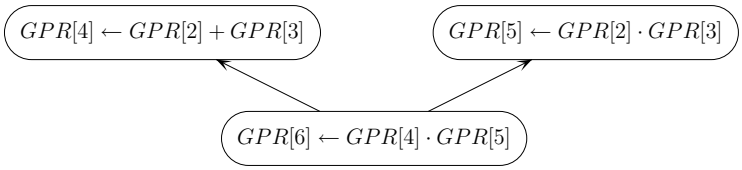


Figure 11.10 A data-flow graph relating to the operation $(a+b)\cdot(a\cdot b)$.

behaviour is data-dependent, we can only really use heuristics unless we have access to some profiling information. For example, an easy heuristic is to mimic fixed branch prediction techniques within processors that guess backward branches are always taken and forward branches are never taken. Once we apply our heuristics, we effectively weight edges in the CFG with information which predicts how likely it is that control-flow will pass along that edge; from this we can guess at the dominante traces by simply favouring basic blocks with highly weighted edges.

11.5.4 Scheduling Instructions

Within a basic block we deal with ordering instructions using a **Data-Flow Graph (DFG)** that captures information about dependencies. Nodes in the DFG represent machine instructions, edges between the nodes represent data-flow: if there is an edge from node X to node Y, X is dependent on a value that Y produces. Using the DFG we can prevent orderings that would cause the program to be invalid: roughly speaking, the edges in the graph put limits on the ordering because if there is an edge from X to Y, X cannot be placed before Y. As well as the explicit dependencies in the DFG, we also need to take care to accommodate implicit dependencies. For example we probably need to preserve the order of accesses to memory.

Consider a contrived example that computes $(a+b)\cdot(a\cdot b)$ using the short intermediate program

$$\begin{aligned} t_0 &\leftarrow a+b \\ t_1 &\leftarrow a\cdot b \\ t_2 &\leftarrow t_0\cdot t_1 \end{aligned}$$

If we assume the symbols a, b, t_0, t_1 and t_2 are allocated the registers $GPR[2]$, $GPR[3]$, $GPR[4]$, $GPR[5]$ and $GPR[6]$, then this can clearly be implemented by the program fragment

$$\begin{aligned} GPR[4] &\leftarrow GPR[2]+GPR[3] \\ GPR[5] &\leftarrow GPR[2]\cdot GPR[3] \\ GPR[6] &\leftarrow GPR[4]\cdot GPR[5] \end{aligned}$$

These instructions produce a small DFG as shown in Figure 11.10. The obvious thing we can take from the graph straight away is that since there is no edge between instructions #0 and #1 (that generate the symbolic values t_0 and t_1 held in $GPR[4]$ and $GPR[5]$), we could execute them in either order and still get the same result. The question is, why would we favour one order over the other ? Imagine we run the program on a superscalar processor (where our aim is to issue one instruction or more per-cycle) with a single PE able to perform multiplication, and that addition and multiplication have latencies of one and two cycles respectively. Executing instruction #0 before #1 means that instruction #2 will stall because the multiplier will be busy executing instruction #1. However, executing instruction #1 before #0 means that by the time we need the multiplier for instruction #2, it is free. Clearly this is a simplistic example, but more or less the same reasoning can be applied to larger programs (and hence larger data-flow graphs).

The DFG can also be useful in determining when we can fill a delay slot with useful instructions. Recall that a branch delay slot is a (small) window of instructions after a branch that, depending on the ISA, are executed whether or not the branch is taken; this is often a side effect of decisions made when implementing the pipelining scheme. A similar but slightly different concept exists in the context of load instructions. We discussed an informal approach of how to fill a delay slot in Chapter 9; using the DFG we can be more specific about how this works. Consider extending our example from above into the intermediate program

$$
\begin{array}{l}
\quad t_0 \leftarrow a+b \\
\quad t_1 \leftarrow a \cdot b \\
\quad t_2 \leftarrow t_0 \cdot t_1 \\
\quad \text{if } t_0 = 0, PC \leftarrow l_1 \\
\hline
\quad NOP \\
\quad NOP \\
\hline
l_0 : \ldots \\
\quad \ldots \\
l_1 : \ldots
\end{array}
$$

where the NOP instructions represent the branch delay slot and the label l_0 marks where the branch actually completes execution. Using the same register allocation as before, we arrive at the program fragment

$$
\begin{array}{l}
\quad GPR[4] \leftarrow GPR[2] + GPR[3] \\
\quad GPR[5] \leftarrow GPR[2] \cdot GPR[3] \\
\quad GPR[6] \leftarrow GPR[4] \cdot GPR[5] \\
\quad \text{if } GPR[4] = 0, PC \leftarrow l_1 \\
\hline
\quad NOP \\
\quad NOP \\
\hline
l_0 : \ldots \\
\quad \ldots \\
l_1 : \ldots
\end{array}
$$

But looking at the DFG, the branch clearly does not depend on instructions #1 or #2 so we could reasonably re-order the instructions and produce

$$
\begin{array}{l}
\quad GPR[4] \leftarrow GPR[2] + GPR[3] \\
\quad \text{if } GPR[4] = 0, PC \leftarrow l_1 \\
\hline
\quad GPR[5] \leftarrow GPR[2] \cdot GPR[3] \\
\quad GPR[6] \leftarrow GPR[4] \cdot GPR[5] \\
\hline
l_0 : \ldots \\
\quad \ldots \\
l_1 : \ldots
\end{array}
$$

This removes the need for padding with NOPs: instructions #1 and #2 will still be performed whether or not the branch is executed and as a result, the program fragment is marginally smaller and more efficient.

11.6 "Template" Code Generation for High-Level Statements

The process of instruction selection and scheduling as a means of converting an intermediate program into a machine-level program demands that first of all we translate a high-level program into an intermediate representation. This is a little beyond the scope of our discussion. Instead, to make it more clear how the translation process *could* work we will consider a simplification: imagine we want to translate a "generalised" high-level language construct (e.g., a `for` loop) into assembly language, how could we do this ? At a very simplistic level, one can imagine this translation being performed using a number of templates which the compiler matches against cases in the source program and fills in to produce an equivalent target program. Of course this is an over simplification of how modern compilers operate, but acts as a good starting point and analogy between how a human might translate programs written in C into programs written in assembly language.

As such, Figure 11.11 details example conditional or `if` statements; Figure 11.12 details a number of loop constructs, for example `do`, `while` and `for` loops; and Figure 11.13 details two strategies for multi-way branch or `switch` statements. The left-hand side of each shows a fragment of C program (i.e., the feature in the source program) and the right-hand side shows an equivalent MIPS32 assembly language template; all use continuation dots extensively to denote arbitrary other statements. Although there are potentially alternative translations depending on the exact context, each example should go some way to demonstrating that even more complicated C statements can easily be decomposed into basic steps that map onto the processor ISA. Additionally, this decomposition helps to understand the relative merits and costs of C statements in terms of both code footprint and execution time.

```
if( x == y )
{
  ...
}
else
{
  ...
}
```

(a) Standard `if` conditional.

```
.text

    ...             # put x into GPR[2]
    ...             # put y into GPR[3]
    bne  $2, $3, l0
    ...             # if   body
    j    l1
l0: ...             # else body
l1:
```

(b) Assembly language translation.

```
if     ( x == y )
{
  ...
}
else if( x == z )
{
  ...
}
else
{
  ...
}
```

(c) Sequenced-`if` conditional.

```
.text

    ...             # put x into GPR[2]
    ...             # put y into GPR[3]
    bne  $2, $3, l0
    ...             # if   body 1
    j    l2
l0: ...             # put x into GPR[2]
    ...             # put z into GPR[4]
    bne  $2, $4, l1
    ...             # if   body 2
    j    l2
l1  ...             # else body
l2:
```

(d) Assembly language translation.

```
if( x == y )
{
  if( x == z )
  {
    ...
  }
  else
  {
    ...
  }
}
else
{
  ...
}
```

(e) Nested-`if` conditional.

```
.text

    ...             # put x into GPR[2]
    ...             # put y into GPR[3]
    bne  $2, $3, l0
    ...             # put x into GPR[2]
    ...             # put z into GPR[4]
    bne  $2, $4, l2
    ...             # if  body  (inner)
    j    l3
l2: ...             # else body (inner)
l3: j    l1
l0: ...             # else body (outer)
l1: ...
```

(f) Assembly language translation.

Figure 11.11 C code translation examples (conditional statements).

```
do
{
  ...
}
while( x < y );
```

(a) Standard `do` loop.

```
.text

10: ...             # do body
    ...             # put x into GPR[2]
    ...             # put y into GPR[3]
    sub  $4, $2, $3
    bltz $4, 10
    ...
```

(b) Assembly language translation.

```
while( x < y )
{
  ...
}
```

(c) Standard `while` loop.

```
.text

10: ...             # put x into GPR[2]
    ...             # put y into GPR[3]
    sub  $4, $2, $3
    bgez $4, 11
    ...             # while body
    j    10
11: ...
```

(d) Assembly language translation.

```
for( x = 0; x < y; x++ )
{
  ...
}
```

(e) Standard `for` loop.

```
.text

    addi $2, $0,  0 # put x into GPR[2]
10: ...             # put y into GPR[3]
    sub  $4, $2, $3
    bgez $4, 11
    ...             # for body
    addi $2, $2,  1
    j    10
    ...
```

(f) Assembly language translation.

Figure 11.12 C code translation examples (loop statements).

11.6.1 Conditional Statements

The first set of fragments in Figure 11.11 deal with various forms of `if` statements; although terminology might differ we refer to the case where there is an `else if` clause as a **sequenced-**`if`, in the sense that the conditions are tested in sequence, and the case where one `if` statement is placed within another as a **nested-**`if`. Consider Figure 11.11a as an example. The C fragment tests whether the value `x` equals the value `y`; if it does, then the true case (i.e., statements represented by the top set of

```
switch( x )
{
  case   0 :
    ...
    break;
  case  50 :
    ...
    break;
  case 100 :
    ...
    break;
}
```

(a) Standard `switch` statement.

```
.text

    ...              # put x into GPR[2]
    addi $3, $0,   0
    bne  $2, $3, l1
    ...              # case body (0)
    j    l3          # break
l1: addi $3, $0,  50
    bne  $2, $3, l2
    ...              # case body (1)
    j    l3          # break
l2: addi $3, $0, 100
    bne  $2, $3,  l3
    ...              # case body (2)
    j    l3          # break
l3: ...
```

(b) Sparse assembly language translation.

```
switch( x )
{
  case 0 :
    ...
    break;
  case 1 :
    ...
    break;
  case 2 :
    ...
    break;
}
```

(c) Standard `switch` statement.

```
.data

jt: .word l1
    .word l2
    .word l3

.text

    ...              # put x into GPR[2]
    addi $3, $0,  2  # scale x
    sllv $4, $2, $3
    lw   $5, jt($4)  # load address
    jr   $5          # jump to address
l1: ...              # case body (0)
    j    l4          # break
l2: ...              # case body (1)
    j    l4          # break
l3: ...              # case body (2)
    j    l4          # break
l4: ...
```

(d) Dense assembly language translation.

Figure 11.13 C code translation examples (multi-way branch statements).

continuation dots) is executed, otherwise the false case (i.e., statements represented by the bottom set of continuation dots) is executed. Of course, the values `x` and `y` might be the results of more complex expressions: in this case we simply translate the expressions into a sequence of instructions which generate the result into the register allocated to `x` or `y`. Also, the test might be something other than that of equality: in this case we simply utilise a different instruction from the processor ISA to invoke the corresponding branch. Either way, we can translate the frag-

ment into the commented assembly language code shown in Figure 11.11b. First we generate `x` or `y` (be they simple variables or expressions) into *GPR*[2] and *GPR*[3] (of course these registers might differ depending on the allocation). Then we test *GPR*[2] against *GPR*[3], if they are not equal, then control is transferred to label `l0` which marks the false case (i.e., code for the bottom set of continuation dots) in the original fragment. After execution of these statements, control will drop through to the label `l1` which marks the end of the `if` statement. If *GPR*[2] and *GPR*[3] are equal, then control drops through from the branch instruction to the code for the true case (i.e., code for the top set of continuation dots). This is followed by an unconditional branch to label `l1` which skips over code for the false case and concludes the `if` statement.

Translation of the sequenced-`if` in Figure 11.11c follows a similar pattern. The difference is that instead of a single true case, there are many true cases which need to be tested using dedicated instructions before finally branching to the default where none of the tests are satisfied. So in this example we first test whether `x` and `y`, held in *GPR*[2] and *GPR*[3], are equal. If they are, control drops through to the first true case (i.e., statements represented by the top set of continuation dots) which is terminated with an unconditional branch to the label `l2` which concludes the `if` statement. If they are not equal, control is transferred to label `l0` which marks the second test of whether `x` and `z`, held in *GPR*[2] and *GPR*[4], are equal. Notice the slight subtlety here in that depending on whether `x` is a variable or an expression or even a function call, we might need to reevaluate it before the test. Either way, execution of the test either transfers control to label `l1`, which marks the false case, or drops through to execute the second true case which is again terminated by a branch to label `l2`.

The nested-`if` in Figure 11.11e is even easier still. Essentially we simply have the second (inner) `if` statement within the true case of the first (outer) `if` statement. So to translate the fragment into the assembly language in Figure 11.11f we take the basic template detailed in Figure 11.11b and fill in the true case with another instance of the same template (with labels and so on renamed).

11.6.2 Loop Statements

The second set of fragments in Figure 11.12 deal with various forms of structured loop statement. The `do` loop in Figure 11.12a, and translated in Figure 11.12b, is perhaps the easiest to understand. Execution immediately enters the loop body at label `l0` meaning the loop is always executed at least once. After completing execution of the loop body we need to test the condition to see if another iteration is required. In this case, we place `x` and `y` into *GPR*[2] and *GPR*[3] and then test them against each other: if `x` is less than `y`, then the branch instruction transfers control back to label `l0` so the loop body is executed again.

The `while` loop in Figure 11.12c, and translated in Figure 11.12d, operates roughly in the opposite order; the loop condition is tested before execution of the

loop body rather than afterwards. Thus execution immediately starts at label `l0` where, in this case, `x` and `y` are placed into *GPR*[2] and *GPR*[3] and then test them against each other: if `x` is not less than `y`, then control is transferred to label `l1` to exit the loop. Otherwise the loop body is executed, a sequence which terminates with an unconditional branch back to `l0` to potentially perform another iteration depending on the condition.

A `for` loop is simply a `while` loop with some extra embellishment. To see how translation is achieved it might be useful to consider that the `for` loop

```
for( x = 0; x < y; x++ ) {
  ...
}
```

can be rewritten as the `while` loop

```
x = 0;

while( x < y ) {
  ...
  x++;
}
```

The translation in Figure 11.12f follows exactly this pattern. Roughly, it adds to the previous `while` loop example by placing the initialisation and increment statements in appropriate positions before the loop test and after the loop body. As an aside, notice that the `while` loop

```
while( x ) {
  ...
}
```

can be written as the `do` loop

```
if( x ) {
  do {
    ...
  }
  while( x );
}
```

This might seem pointless from the view of a human programmer. However, for a compiler, the fact that we can use a single, canonical representation of `for`, `while` and `do` loops means less "special cases": we can represent all three in the same way and hence apply just one set of analysis and transformation methods rather than three.

11.6.3 Multi-way Branch Statements

The third and final set of fragments in Figure 11.13 deal with the multi-way branch. The basic idea is that depending on some value, in this case `x`, control should be

transferred to one of many target addresses; this contrasts with a normal branch where control is transferred to one of two target addresses. The C `switch` statement is an example of a multi-way branch: control is transferred to a `case` block marked with the value corresponding to `x`. With a `switch` statement, the values labelling each `case` statement are required to be constant (i.e., known at compile-time). This allows the use of a translation strategy which best suits the values themselves. Interestingly, a compiler might use the first strategy, based on a sequenced-`if`, where the values are *sparsely* distributed (e.g., $0, 50, 100, \ldots$), and the second, termed a **branch table**, where the values are *densely* packed together (e.g., $0, 1, 2, \ldots$).

Figure 11.13a and Figure 11.13b demonstrate the sequenced-`if` technique. We simply test `x` against each of the cases in turn and if a match occurs, execute the corresponding statements which are followed by an unconditional branch to label `l3` to implement the `break`. Since this is based on the sequenced-`if`, implementation of the `default` case, which should match if no others do, is simple since this is analogous to the `else` case in Figure 11.11c and Figure 11.11d.

Figure 11.13c and Figure 11.13d demonstrate the branch table strategy. Essentially the idea is to take `x` and use it as an offset into a table defined at the label `jt`; each entry in the table tells us where to branch to for that `x`. For example, entry zero in the table contains the address of label `l1`, entry one in the table contains the address of label `l2` and entry two contains the address of label `l3`. The strategy is to take `x` and scale it into an offset; we can then load the appropriate address and then branch to it using a `jr` instruction. First we generate `x` into $GPR[2]$ and then scale it, multiplying by four so that the result in $GPR[5]$ gives a valid offset into a table of 32-bit addresses. Then we load the address, using the `jt` label as a base and the scaled value in $GPR[5]$ as an offset, and branch to the result. This method is more efficient: no matter what the value of `x`, we always perform the same sequence of instructions (i.e., do not need to test for cases that will not match). However, it makes the assumption that the table offset can be generated in a valid manner from `x`. For example, if there is no case for `x` equal to 1, there is a gap in the table we need to cope with. Additionally, it is harder to deal with the `default` case using this method.

11.7 "Template" Code Generation for High-Level Function Calls

In Chapter 5 we gave a very basic overview for how function calls are implemented; we focused mainly on the concept of the branching from the **caller**, the part that makes the function call, to the **callee**, the function being called. To create a more complete mechanism, we need to consider three main issues: passing arguments from the caller to the callee function; creating local variables for the function; and returning results from the callee to the caller function.

First, however, we need to consider in more detail how the stack is controlled. Typically, MIPS32 assemblers usually allow `$sp` as a register alias for the stack

pointer; as such we can implement PUSH and POP operations on 32-bit words using two instructions

```
...
addi $sp, $sp, -4 # make space
sw   $2, 0($sp)   # push from GPR[2]
...
lw   $2, 0($sp)   # pop  into GPR[2]
addi $sp, $sp,  4 # free space
...
```

Notice that once we have pushed some values onto the stack, we can inspect their values without actually popping them: the stack is simply memory after all. This is sometimes termed peeking at the stack content. Imagine we again initially set $x = 3$ and assume the memory is zeroed, i.e., that $MEM[0\ldots2] = (0,0,0)$. Then we push three values, abstractly named a, b and c, onto the stack

$$\begin{aligned} x = \text{PUSH}(x,a) &\rightarrow x = 2,\ \ MEM[0\ldots2] = (0,0,a) \\ x = \text{PUSH}(x,b) &\rightarrow x = 1,\ \ MEM[0\ldots2] = (0,b,a) \\ x = \text{PUSH}(x,c) &\rightarrow x = 0,\ \ MEM[0\ldots2] = (c,b,a). \end{aligned}$$

We can access a,b and c as $MEM[2]$, $MEM[1]$ and $MEM[0]$. Often it is often more attractive to access the stack content relative to SP; thus, with the stack in the final state, we access a,b and c as $MEM[SP+2]$, $MEM[SP+1]$ and $MEM[SP+0]$. This is sometimes called **stack relative addressing**. The problem with this approach is that SP changes when values are pushed and popped, so accessing say $MEM[SP+1]$ gives a different result depending on what state we are in: after the first push it accesses a, after the second it accesses b and after the final push it accesses c. So to correctly use stack relative addressing we need to be clear what state the stack is in so as to give the right offset from SP.

One use of the stack is as a means of implementing function calls; we will look at this in detail later. A simpler example is the occasional need to store a value somewhere temporarily. Imagine for example we have a code sequence where some 32-bit sized variable a is stored in $GPR[2]$ and set to the value 10. There is only a limited number of registers so sometime later in the sequence we run out of spare ones: all the registers are all full of values we need. To continue, we need to free up one (or more) registers; the basic idea here is to find part of the code sequence which does not use a, for example, and to backup or **spill** the value into memory temporarily. This frees the register $GPR[2]$ for use to use without losing the value of a since we can simply restore it from memory again later when we need it. The stack is ideal for this task and we can implement the spill using code similar to the following:

```
...
addi $2,  $2,  10 # set a to 10
...
addi $sp, $sp, -4 # make a space
sw   $2, 0($sp)   # push a from GPR[2]
...               # now free to use GPR[2] for something else
lw   $2, 0($sp)   # pop  a into GPR[2]
```

```
addi $sp, $sp,  4 # free a space
...
```

Finally, note that it is quite common to push or pop many values to or from the stack at once. In this case, it is quite inefficient to decrement or increment *SP* for each push or pop. It is better to simply perform a single decrement or increment and then store or load the values with appropriate offsets. For example, pushing and popping the 32-bit sized variables *a*, *b* and *c* from and to *GPR*[2], *GPR*[3] *GPR*[4] might, in the naive case, look something like

```
...
addi $sp, $sp,  -4 # make a space
sw   $2, 0($sp)    # push a from GPR[2]
addi $sp, $sp,  -4 # make b space
sw   $3, 0($sp)    # push b from GPR[3]
addi $sp, $sp,  -4 # make c space
sw   $4, 0($sp)    # push c from GPR[4]
...
lw   $4, 0($sp)    # pop  c into GPR[4]
addi $sp, $sp,   4 # free c space
lw   $3, 0($sp)    # pop  b into GPR[3]
addi $sp, $sp,   4 # free b space
lw   $2, 0($sp)    # pop  a into GPR[2]
addi $sp, $sp,   4 # free a space
...
```

but clearly we can shorten this significantly to

```
...
addi $sp, $sp, -12 # make a, b and c space
sw   $2, 8($sp)    # push a from GPR[2]
sw   $3, 4($sp)    # push b from GPR[3]
sw   $4, 0($sp)    # push c from GPR[4]
...
lw   $4, 0($sp)    # pop  c into GPR[4]
lw   $3, 4($sp)    # pop  b into GPR[3]
lw   $2, 8($sp)    # pop  a into GPR[2]
addi $sp, $sp,  12 # free a, b and c space
...
```

11.7.1 Basic Stack Frames

The most common way to solve the highlighted problems is to use the stack described previously. When a function is called, it is allocated space on the stack called a **stack frame**. When the function returns, the stack frame is destroyed. The stack frame is, in some sense, shared between the caller and the callee and offers a way to communicate values between the two: the arguments to the callee are written by the caller and read by the callee; the return address is written by the caller and read by the callee; the return value is written by the callee and read by the caller. Typically,

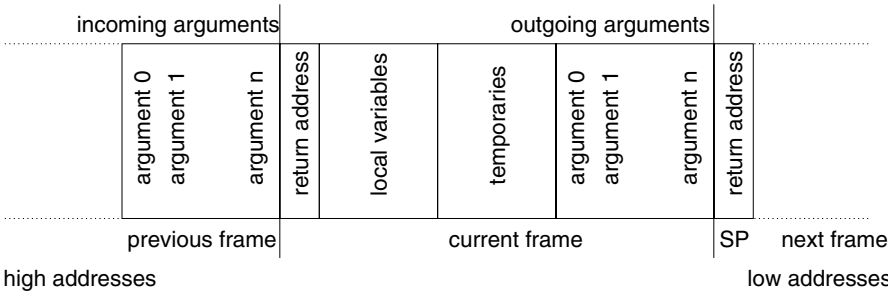


Figure 11.14 A simple stack frame.

```
caller: push arguments
        call callee
        pop  return value
        pop  arguments
        ...

callee: push return address
        push space for locals
        ...
        pop  space for locals
        pop  return address
        push return value
        return to caller
```

Figure 11.15 A simple calling convention between caller and callee.

we name the code sequences that allocate and destroy the stack frame the **function prologue** and **function epilogue**. The specification of the function prologue and epilogue forms the function **calling convention**. Figure 11.15 demonstrates a simple calling convention written as pseudo-code.

This fairly vague description demands an example to make things more concrete. Consider the pseudo-assembly language in Figure 11.15 which implements a simple calling convention. The caller pushes arguments onto the stack then makes the function call; in MIPS32 this is performed with the jump and link or `jal` instruction which jumps to the callee address and stores the return address in register `$ra$`. The callee immediately stores this return address on the stack and allocates space for any local variables that might need to be spilt to memory. When it has finished, it frees the space allocated for local variables then pops the return address from the stack, pushes any return value and branches back to the caller. The caller finalised the function call by retrieving the return value and the freeing space on the stack previously allocated for arguments. Figure 11.14 shows the general layout of stack frames managed by this calling convention; one can immediately see how the

```
.globl main

.text

main: ...                   #              initialise n,  n is in $t2
      ...                   # allocate and initialise A, &A is in $t0
      ...                   # allocate and initialise B, &B is in $t1

      addi    $sp, $sp,  -4 # make &A space
      sw      $t0,  0($sp)  # push &A
      addi    $sp, $sp,  -4 # make &B space
      sw      $t1,  0($sp)  # push &B
      addi    $sp, $sp,  -4 # make  n space
      sw      $t2,  0($sp)  # push  n
      jal     dot           # call dot function
      nop                   # delay slot
      nop                   # delay slot
      lw      $t3,  0($sp)  # pop  rv
      addi    $sp, $sp,   4 # free rv space
      addi    $sp, $sp,   4 # free  n space
      addi    $sp, $sp,   4 # free &B space
      addi    $sp, $sp,   4 # free &A space
                            # dot result in $t3
      addi    $v0, $0,   10
      syscall               # exit

dot:  addi    $sp, $sp,  -4 # make ra space
      sw      $ra,  0($sp)  # push ra
      addi    $sp, $sp,  -4 # make  i space
      addi    $sp, $sp,  -4 # make  a space

      lw      $t0, 20($sp)  # load X from stack
      lw      $t1, 16($sp)  # load Y from stack
      lw      $t2, 12($sp)  # load n from stack
      addi    $t3, $0,    0 # set a to 0
      addi    $t4, $0,    0 # set i to 0
l0:   sub     $t5, $t2, $t4
      blez    $t5, l1       # goto l1 if n <= i
      addi    $t5, $0,    4
      mul     $t5, $t5, $t4
      add     $t6, $t5, $t0
      lw      $t7,  0($t6)  # load X[i]
      add     $t6, $t5, $t1
      lw      $t8,  0($t6)  # load Y[i]
      mul     $t9, $t7, $t8
      add     $t3, $t3, $t9 # set a = a + ( X[i] * Y[i] )
      addi    $t4, $t4,   1 # set i to i + 1
      j       l0            # goto l0
l1:
      addi    $sp, $sp,   4 # free  a space
      addi    $sp, $sp,   4 # free  i space
      lw      $ra,  0($sp)  # pop  ra
      addi    $sp, $sp,   4 # free ra space
      addi    $sp, $sp,  -4 # make rv space
      sw      $t3,  0($sp)  # push rv
      jr      $ra           # return
```

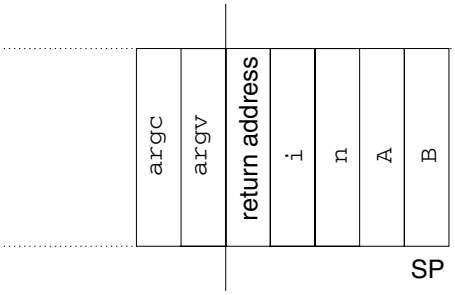
Listing 11.2 MIPS32 implementation of dot product function using a basic calling convention.

frames are shared between caller and callee and how the stack grows downward in memory.

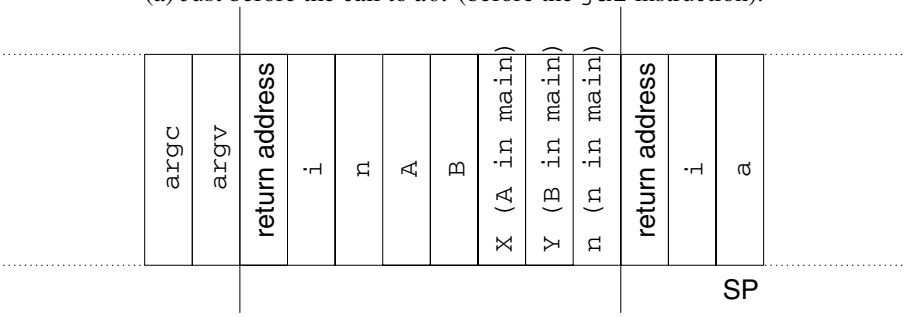
Listing 11.2 is an implementation of the example from Chapter 10, in Listing 10.1 and Listing 10.2, which performs a dot product calculation. Notice that following the basic introduction, we are now using aliased register names because we will start assigning various registers specific roles. Plus or minus some minor details, the program implements the steps from Figure 11.15 exactly. Starting at the entry point `main`, we first assume that the variables are initialised with their values held in `$t0`, `$t1` and `$t2` respectively; *A* and *B* are pointers so their values are really addresses. At this point the stack frame resembles Figure 11.16a in the sense that the `main` function, when called, was passed the arguments `argc` and `argv` and allocated space on the stack for the local variables *i*, *n*, *A* and *B*. The meat of the example code starts when it calls the `dot` function. First it pushes the values of *A*, *B* and *n* onto the stack as arguments to `dot`. It then invokes the function as the caller; we will come back to what `dot`, the callee does in a while. Notice that the code is manually padded with `nop` instructions that fill the `jal` delay slot. This ensures that no rogue instructions are executed. When control returns to `main` from `dot` the stack frame resembles Figure 11.16c; notice that the return value is at the top of the stack. The code retrieves this value into `$t3` and then frees the stack space previously allocated for the arguments to complete the call.

From the point of view of the callee, the steps are again a replication of the example. The function `dot` starts by performing the function prologue in saving the return address, held in `$ra`, onto the stack and allocating some space for the local variables *i* and *a*. This completes the stack frame for `dot`, it resembles Figure 11.16b. Notice that `dot` starts by retrieving the incoming arguments *X*, *Y* and *n* using loads relative to *SP*. For example, the argument *Y* is located at an address (counting in 32-bit words) specified by *SP* plus the number of local variables and return address we pushed, plus the index of *Y* in the argument list. That is, the value of *Y* is located at $SP+2+1+1 = SP+4$ words or $SP+16$ bytes. Using a similar logic, *X* is located at $SP+2+1+2 = SP+5$ words or $SP+20$ bytes. Once the arguments are available, the actual function body is somewhat simple: we execute a loop which loads values of X_i and Y_i to compute the accumulated value *a*. When the loop terminates, we need to perform the function prologue. First, it frees the stack space previously allocated for local variables *i* and *a*. Then it retrieves the return address from the stack, pushes the return value *a* and returns using the `jr` instruction which jumps to the address stored in `$ra` i.e., the return address.

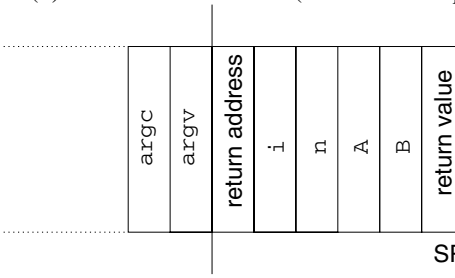
This approach provides a working function call mechanism but we can easily identify some improvements to the implementation. Firstly, inside `dot` we allocate space on the stack for local variables *i* and *a* in case we need to spill them to memory. However, this never actually happens; therefore the space and extra instructions are redundant and we can safely remove this feature. Secondly, the caller uses a pattern we previously identified as inefficient in using repeated decrements and increments to push and pop arguments. By aggregating these into one single update of *SP* we further improve the program. Both these improvements are implemented in Listing 11.3.



(a) Just before the call to *dot* (before the `jal` instruction).



(b) Inside the call to *dot* (after function prologue).



(c) Just after return from *dot* (after `jal` instruction).

Figure 11.16 Dynamic behaviour of the stack while calling functions.

11.7.2 Advanced Stack Frames

There are some major problems with this simple calling convention. The first problem is one of efficiency. On one hand, structured programming demands modularity through small, easy to manage functions; on the other hand, the actual function call mechanism represents overhead compared to performing useful computation. Much of this overhead is introduced by management of the stack frame. To partly reduce this cost we can communicate arguments and return values between caller and callee

```
1  .globl main
2
3  .text
4
5  main: ...                    #              initialise n,  n is in $t2
6        ...                    # allocate and initialise A, &A is in $t0
7        ...                    # allocate and initialise B, &B is in $t1
8
9        addi    $sp, $sp, -12 # make &A space
10                              # make &B space
11                              # make  n space
12       sw      $t0,  8($sp)  # push &A
13       sw      $t1,  4($sp)  # push &B
14       sw      $t2,  0($sp)  # push  n
15       jal     dot           # call dot function
16       nop                   # delay slot
17       nop                   # delay slot
18       lw      $t3,  0($sp)  # pop  rv
19       addi    $sp, $sp,   4 # free rv space
20                              # free  n space
21                              # free &B space
22       addi    $sp, $sp,  12 # free &A space
23                              # dot result in $t3
24       addi    $v0, $0,   10
25       syscall               # exit
26
27 dot:  lw      $t0,  8($sp)  # load X from stack
28       lw      $t1,  4($sp)  # load Y from stack
29       lw      $t2,  0($sp)  # load n from stack
30       addi    $t3, $0,    0 # set a to 0
31       addi    $t4, $0,    0 # set i to 0
32 l0:   sub     $t5, $t2, $t4
33       blez    $t5, l1       # goto l1 if n <= i
34       addi    $t5, $0,    4
35       mul     $t5, $t5, $t4
36       add     $t6, $t5, $t0
37       lw      $t7,  0($t6)  # load X[i]
38       add     $t6, $t5, $t1
39       lw      $t8,  0($t6)  # load Y[i]
40       mul     $t9, $t7, $t8
41       add     $t3, $t3, $t9 # set a = a + ( X[i] * Y[i] )
42       addi    $t4, $t4,   1 # set i to i + 1
43       j       l0            # goto l0
44 l1:
45       addi    $sp, $sp,  -4 # make rv space
46       sw      $t3,  0($sp)  # push rv
47       jr      $ra           # return
```

Listing 11.3 MIPS32 implementation of dot product function still using a basic calling convention but with some optimisations.

via a **pass-by-register** convention rather than the **pass-by-stack** convention used so far. The basic concept is simple: rather than the caller pushing arguments onto the stack only for the callee to later load them into registers ready for use, the caller just places them in registers to start with. These registers need to be saved and restored just like normal registers if overwritten by subsequent function calls. It is important that the caller and callee agree on a standard for the passing convention, otherwise neither will know where the required values are. For example, the MIPS32 calling convention specifies that registers $GPR[2\ldots3]$ or `$2` and `$3` (or `$v0` and `$v1`

```
1  .globl main
2
3  .text
4
5  main: ...                  #              initialise n,  n is in $t2
6        ...                  # allocate and initialise A, &A is in $t0
7        ...                  # allocate and initialise B, &B is in $t1
8
9        add     $a0, $t0, $0  # set argument 0 to &A
10       add     $a1, $t1, $0  # set argument 1 to &B
11       add     $a2, $t2, $0  # set argument 2 to  n
12       jal     dot           # call dot function
13       nop                   # delay slot
14       nop                   # delay slot
15                             # dot result in $v0
16       addi    $v0, $0,   10
17       syscall               # exit
18
19 dot:  addi    $v0, $0,    0 # set a to 0
20       addi    $t4, $0,    0 # set i to 0
21 l0:   sub     $t5, $a2, $t4
22       blez    $t5, l1       # goto l1 if n <= i
23       addi    $t5, $0,    4
24       mul     $t5, $t5, $t4
25       add     $t6, $t5, $a0
26       lw      $t7,  0($t6)  # load X[i]
27       add     $t6, $t5, $a1
28       lw      $t8,  0($t6)  # load Y[i]
29       mul     $t9, $t7, $t8
30       add     $v0, $v0, $t9 # set a = a + ( X[i] * Y[i] )
31       addi    $t4, $t4,   1 # set i to i + 1
32       j       l0            # goto l0
33 l1:
34       jr      $ra           # return
```

Listing 11.4 MIPS32 implementation of dot product function still using a more advanced calling convention but with pass-by-register rather than pass-by-stack.

using appropriate aliases) are used to return values from callee to caller and registers $GPR[4\ldots7]$ or `$4` to `$7` (or `$a0` to `$a3` using appropriate aliases) are used to pass arguments from caller to callee. The only remaining issue is that we have a fixed number of registers; clearly not all arguments can be passed this way. Typically, only the first three or four are passed in registers, the rest on the stack. Since functions with a small number of arguments are more common, on average most argument will be in registers.

Listing 11.4 follows on from Listing 11.3 using pass-by-register to further improve the efficiency of our example; simply counting the number of instructions should give some idea of the advantage. More specifically, look at the code for pushing arguments in the caller and retrieving them in the callee. The previous code has essentially disappeared: for example, there is no need for the callee to retrieve the arguments from the stack; they are passed in registers `$a0`, `$a1` and `$a2` and are hence immediately available. Likewise, when returning from the callee to the caller the return value *a* does not need to be pushed onto the stack only to be popped off again straight away. It is allocated register `$v0` throughout and the caller can simply read the value from there when it regains control.

The second problem relates to what happens if the caller function has values stored in registers and those registers are also used by the callee. Clearly we need to save the register content before they are overwritten and restore them afterwards. The act of saving and restoring register is fairly simple since we can again use the stack as previously discussed. However, there are two different ways to decide who performs these operations:

Caller-save In this scheme the caller is responsible for saving and restoring all registers that contain content it needs. The caller does not know which registers will actually be used, so it might be overly pessimistic and save more registers than are strictly required.

Callee-save In this scheme the callee is responsible for saving and restoring all registers that it overwrites. The callee does not know which register values the caller actually needs, so it also might end up being overly pessimistic and saving more registers than strictly required.

Often a combination of the two options is used, for example the MIPS32 calling convention specifies that registers $GPR[8\ldots15]$ and $GPR[24\ldots25]$ are caller-save registers used for temporary values while registers $GPR[16\ldots23]$ are callee-save registers used for temporary values. These can also be accessed using the aliases `$t0` to `$t9` and `$s0` to `$s7` respectively.

The final problem is related to stack offset addressing, specifically that the value of *SP* keeps moving as we perform push and pop operations. This feature means that it can be hard to work out the offset from *SP* required to access a particular element in a stack frame. Furthermore, depending on the actual function this offset can be quite large whereas to fit nicely into our instruction encoding it may be advantageous to keep offsets small. To solve this problem, we introduce a **frame pointer** called FP which is like *SP* except that it remains fixed for the duration of a function invocation, pointing to the base of the stack frame for that invocation. MIPS32 uses $GPR[30]$ or `$fp` to store FP. Since FP is fixed, all offsets into the stack frame are more easily calculated; since it points at the base of the stack frame most accesses to the important entries, such as the function arguments, are efficient since they are a small offset away. The only down side to this approach is that we need to save the FP value when calling subsequent functions; this takes a small amount of extra time and space.

These final problems are solved with the complete, more advanced calling convention pseudo-code shown in Figure 11.18. The additional steps in the caller are the pushing and popping of register values it has responsibility for saving. The additional steps in the callee are the pushing and popping of register values it has responsibility for saving and also management of the frame pointer FP. This calling convention is made use of by taking another look at the problem of computing Fibonacci numbers. This time, instead of generating the first n numbers in the sequence into an array, imagine we want just the n-th number. Furthermore, we are interested in implementing an elegant, recursive solution as shown in Algorithm 11.5. Although it perhaps is not the *most* efficient method we could select, the function is implemented in Listing 11.5 using MIPS32 assembly language.

```
1  main: addi    $a0, $0,   10 # set argument 0 to 10
2        jal     fib           # call fib(10) function
3        nop                   # delay slot
4        nop                   # delay slot
5                              # fib(10) result in $v0
6        addi    $v0, $0,   10
7        syscall               # exit
8
9  fib:  addi    $sp, $sp,  -4 # make ra space
10       sw      $ra,  0($sp)  # push ra
11       addi    $sp, $sp,  -4 # make fp space
12       sw      $fp,  0($sp)  # push fp
13       add     $fp, $sp, $0  # set  fp to sp
14
15       addi    $t0, $0,  0
16       addi    $t1, $0,  1
17       beq     $t0, $a0, l0  # check if n = 0
18       beq     $t1, $a0, l1  # check if n = 1
19
20       addi    $sp, $sp,  -4 # make n space
21       sw      $a0,  0($sp)  # push n
22       addi    $a0, $a0,  -1 # set argument 0 to n - 1
23       jal     fib           # call fib(n-1) function
24       nop                   # delay slot
25       nop                   # delay slot
26                             # fib(n-1) result in $v0
27       lw      $a0,  0($sp)  # pop  n
28       addi    $sp, $sp,   4 # free n space
29
30       addi    $sp, $sp,  -4 # make a space
31       sw      $v0,  0($sp)  # push a
32
33       addi    $a0, $a0,  -2 # set argument 0 to n - 2
34       jal     fib           # call fib(n-2) function
35       nop                   # delay slot
36       nop                   # delay slot
37                             # fib(n-2) result in $v0
38
39       lw      $v1,  0($sp)  # pop  a
40       addi    $sp, $sp,   4 # free a space
41
42       add     $v0, $v0, $v1 # set rv = a + b
43       j       l2
44 l0:   addi    $v0, $0,    0 # set rv =     0
45       j       l2
46 l1:   addi    $v0, $0,    1 # set rv =     1
47 l2:
48       lw      $fp,  0($sp)  # pop  fp
49       addi    $sp, $sp,   4 # free fp space
50       lw      $ra,  0($sp)  # pop  ra
51       addi    $sp, $sp,   4 # free ra space
52       jr      $ra           # return
```

Listing 11.5 MIPS32 implementation of a Fibonacci number generator using advanced calling convention with pass-by-register, caller and callee saved registers and a frame pointer .

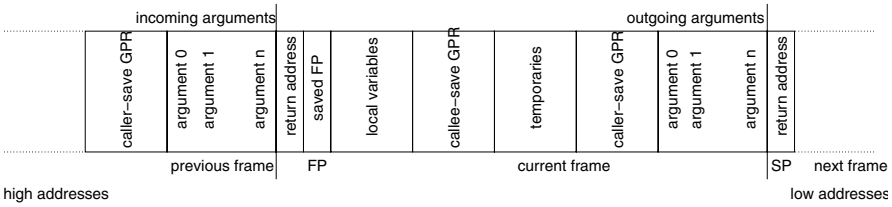


Figure 11.17 A more complete stack frame.

```
caller:
  push caller-save registers
  push arguments
  call
  pop  return value
  pop  arguments
  pop  caller-save registers
  ...

callee:
  push return address
  push saved FP
  set  FP = SP
  push space for locals
  push callee-save registers
  ...
  pop  callee-save registers
  pop  space for locals
  pop  saved FP
  pop  return address
  push return value
  return
```

Figure 11.18 A more complete calling convention between caller and callee.

The function prologue follows the new calling convention by saving the return address and current FP on the stack before updating FP with the current SP value. Although the code does not use this value of FP (since arguments and so on are passed in registers), any stack content can be accessed relative to this fixed address. The code then checks the base cases for the recursion before dropping through to the main body where we need to perform two recursive calls to FIB. Before the first call the code saves the value of n: this call overwrites the register `$a0` that stores n in preparing the new argument $n-1$. After the call completes, we restore n from the stack. Notice that saving n to the stack is not required for the second call since by the time we prepare the argument $n-2$, we no longer need n. However, before the second call we need to save the value of a since the call might destroy this. Although we are not dealing with MIPS32 specified caller-save registers, this is more or less a case of caller-save behaviour: the value of a is being saved by the caller since the

Input: An integer n.
Output: The n-th Fibonacci number.

```
1 FIB(n)
2     if n = 0 then
3         return 1
4     if n = 1 then
5         return 1
6     a ← FIB(n − 1)
7     b ← FIB(n − 2)
8     return a + b
9 end
```

Algorithm 11.5: A recursive algorithm for computing the n-th Fibonacci number.

callee might overwrite it. After the call we restore a, have b computed as the result of the second call and can hence produce their sum and return it via the function epilogue which follows the calling convention by restoring the FP value, retrieving the return address then returning to the caller.

11.8 Further Reading

- A.W. Appel and M. Ginsburg.
  Modern Compiler Implementation in C.
  Cambridge University Press, 2004. ISBN: 0-521-60765-5.
- R.L. Britton.
  MIPS Assembly Language Programming.
  Prentice-Hall, 2003. ISBN: 0-131-42044-5.
- J.R. Levine.
  Linkers and Loaders.
  Morgan-Kaufmann, 2000. ISBN: 1-558-60496-0.
- S.S. Muchnick.
  Advanced Compiler Design and Implementation.
  Morgan-Kaufmann, 1997. ISBN: 1-558-60320-4.
- J. Waldron.
  Introduction to RISC Assembly Language.
  Addison-Wesley, 1998. ISBN: 0-201-39828-1.

11.9 Example Questions

44. Outline the purpose of a stack frame in relation to how function calls work, and describe the purpose of each component in a typical stack frame.

45. A programmer writes the following program in C to find the maximum of two integer values:

```
int max( int x, int y )
{
  if( x > y )
    return x;
  else
    return y;
}

int main( int argc, char* argv[] )
{
  int a = 10;
  int b = 20;

  return max( a, b );
}
```

a. Assuming execution starts by calling the `main` function, describe using a diagram what the stack looks like at **each** of the following stages of execution:
   i. Inside the `main` function, before `max` is called.
   ii. Inside the `max` function.
   iii. Inside the `main` function, after `max` is called.

b. Ignoring the issue of delay slots (i.e., assuming that the assembler managed this automatically), translate the program into MIPS32 assembly language.

46. RISC processors are often designed to support the characteristics of average case C programs.

a. Describe **two** examples where such an approach could be said to have been used in a concrete RISC processor design.
b. Choose a programming language **other than C or C++** (e.g., Java or Python) and imagine you are designing a new RISC processor which will predominately execute programs written in this language. Outline **two** characteristics of such programs which are crucial to performance, and mechanisms you could include in the processor design to support them.

47. Imagine programs written in C require strict array bounds checking due to various security requirements placed on their execution by a MIPS32-based processor. This means that a fragment such as

```
A[ i ] = 0;
```

where *A* is a 1024-element array of 32-bit integers for example, will instead need to be rewritten to include a bounds check:

```
if( i >= 0 && i < 1024 )
  A[ i ] = 0;
else
  abort();
```

Describe **one** way said processor could be improved to cope better with this demand.

Chapter 12
Operating Systems

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– D. Ritchie

Operating systems are like underwear - nobody really wants to look at them.

– B. Joy

Abstract Although programs written for embedded platforms, for example, are often said to execute on "bare metal" (i.e., the unmanaged processor), more usually we write programs that do not execute in isolation or without assistance. More often, they execute on a managed platform which is supported by an **Operating System (OS)**. The OS is a software layer tasked with management of the computer as a whole: it ensures that resources are shared between many executing processes, and performs certain low-level tasks on their behalf. In this chapter the aim is to focus on aspects of OS design and implementation which most directly relate to the hardware and software components the OS manages. We concentrate on UNIX-style concepts and approaches, particularly those which can be supported by SPIM, but the discussion is more generally applicable.

12.1 Introduction

An **Operating System (OS)** is a layer of software that exists to perform various management tasks. At the most basic level, the OS is tasked with loading a program image into memory and providing a context in which it executes as a process; we can identify more advanced tasks using some examples:

- Whether internal or external, events will occur that require the attention of the OS. Examples include some external device that needs to be attended to (e.g., the keyboard has a key press ready to be read), or an executing process that

needs the OS to perform some function on behalf of it (e.g., write output to a display device). As such, the OS needs to provide suitable mechanisms so it can be invoked to deal with such events.

- It is unusual for one process to have dedicated access to the components that constitute a processor. More commonly we would like to have several processes resident in memory, but allow them to coexist by sharing the processor and associated resources having one process active at a given point in time. It is easy to see this in action: a single computer might allow several users to login and execute programs at the same time, or one user might execute a word processor and web-browser at the same time. The OS needs to provide the mechanism by which this is possible.
- Many peripheral devices can be attached to a computer; examples include resources such as display devices, storage devices and so on. Somehow, a process executing on the processor needs to access those devices. To permit some level of portability, access to the devices should be somewhat abstract. For example, if a program needs to know the exact storage device it is accessing, it cannot be executed on a computer with another device. A better approach would be to have the OS provide the program with a generic view of all storage devices and manage the details of access behind the scenes.
- Given that many processes can be sharing the processor, there is a danger that one will influence the execution of another or, worse still, somehow influence the OS itself. For example, you might imagine that if one process accidentally writes values into memory at addresses that hold the instructions for some second process, the corrupted second process probably will not execute as expected. The OS provides mechanisms that protect processes from each other in this respect.
- There is also a chance that two processes sharing the processor will need to access the same resource at the same time. Sometimes this is not possible, for example perhaps the resource is a storage device which cannot read and write data at the same time: while one process is using it to write, another cannot instruct it to read. In this case the OS needs to act as an arbitrator between the processes. That is, and more generally, it needs to manage the resource in order that usage is functionally valid (i.e., works correctly for each process) and fair (e.g., does not indefinitely prevent one process from performing an access).

The problem is, our typical view of an OS is much wider ranging than we would like to investigate here. An "operating system" such as Microsoft Windows, for example, includes a huge range of auxiliary software and features (e.g., any **Graphical User Interface (GUI)**, or software tools such as calculators and text editors) that we would like to ignore.

Adopting a purist perspective, we focus specifically on topics that relate to an OS **kernel**: the central component of the OS that most closely interacts with the hardware devices investigated up until now. We cover roughly three core topics: **event management** which details how the kernel is invoked to deal with different events that can occur, **memory management** which details how the kernel manages memory, the main resource attached to the processor that we previously discussed, and **process management** which details how the kernel manages the execution of

many processes. Roughly speaking the first two topics define the hardware platform which the OS manages on behalf of software that executes according to the third topic. There are clearly many other topics missing from this list; we motivate our choices by the fact that event, process and memory management place direct constraints on and rely upon features in the processor design.

Since SPIM [35], outlined in Appendix A, uses a simulation=based approach to executing programs, it does not model some features of a MIPS32-based processor which are required to support a full kernel; for example, it does not fully model the memory hierarchy. As a result, we adopt a practical approach to the topics it can support (e.g., we develop a real exception handler) but overview those it does not at a higher-level (e.g., virtual memory).

12.2 The Hardware/Software Interface

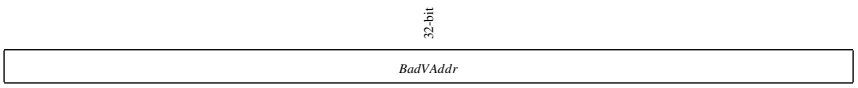
If the OS is tasked with managing the interface between hardware and software, this interface needs to include the ability for the kernel to inspect and control some aspects of how the processor operates. Before we discuss how the kernel implements said interface, we need to investigate the mechanism by which that inspection and control is exercised. For MIPS32 at least, this is defined in the **Privileged Resource Architecture (PRA)** which in some sense defines how the processor operates in "managed" mode and contrasts with the ISA which defines how the processor operates in "bare metal" mode.

12.2.1 MIPS32 Co-processor Registers

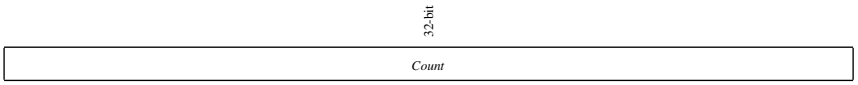
The MIPS32 ISA includes an interface to co-processor devices. One might view these as physically separate from the processor; for our purposes, however, we can view them simply as collections of special-purpose registers. We denote these in a similar way to the general-purpose register file so that $CP0[i]$ is the i-th register within co-processor zero, or $CP1[j]$ is the j-th register in co-processor one. Access to the registers is achieved by copying values to and from the general-purpose register file using dedicated instructions. For example, the `mtc0` and `mfc0` copy values to and from co-processor zero. As a result, we might write the assembly language fragment

```
mfc0 $t0, $0 ; move CP0[0] into GPR[8]
...          ; perform some processing of GPR[8]
mtc0 $t0, $0 ; move GPR[8] into CP0[0]
```

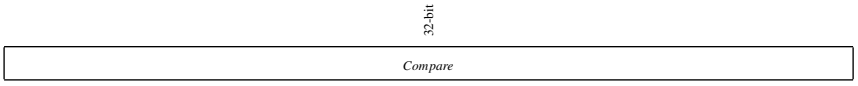
to read the value of $CP0[0]$, perform some change to the value and then write this back to update the co-processor. The idea is that in reality this would represent communication between the processor and co-processor, potentially prompting the co-processor to perform some action on behalf of the processor.



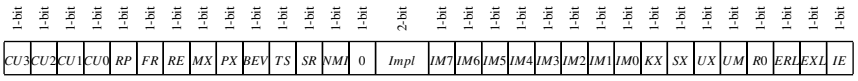
(a) The *BadVAddr* register, *CP*0[8].



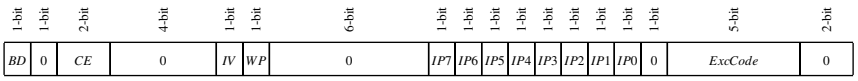
(b) The *Count* register, *CP*0[9].



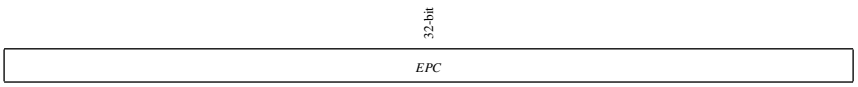
(c) The *Compare* register, *CP*0[11].


1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 2-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit
CU3 CU2 CU1 CU0 RP FR RE MX PX BEV TS SR NMI 0 Impl IM7 IM6 IM5 IM4 IM3 IM2 IM1 IM0 KX SX UX UM R0 ERL EXL IE

(d) The *Status* register, *CP*0[12].


1-bit 1-bit 2-bit 4-bit 1-bit 1-bit 6-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 1-bit 5-bit 2-bit
BD 0 CE 0 IV WP 0 IP7 IP6 IP5 IP4 IP3 IP2 IP1 IP0 0 ExcCode 0

(e) The *Cause* register, *CP*0[13].



(f) The *EPC* register, *CP*0[14].

(g) The *Config* register, *CP*0[16].

Figure 12.1 The sub-set of MIPS32 co-processor zero registers from Table 12.1 described as bit-fields.

Register Number	Register Name	Description
8	*BadVAddr*	Address for most recent address-related exception.
9	*Count*	Cycle accurate time-stamp counter.
11	*Compare*	Timer interrupt control.
12	*Status*	Processor status.
13	*Cause*	Details of exception meaning.
14	*EPC*	Exception Program Counter (EPC).
16	*Config*	Processor configuration.

Table 12.1 A table describing a sub-set of MIPS32 co-processor zero registers.

Although they take different approaches to control of their behaviour, processors need somewhere to store configuration and some mechanism to inspect and update it. Although it supports up to three co-processors, a MIPS32-based processor dedicates co-processor zero for this role. Of the thirty two or so registers in $CP0$, some are optionally defined and only a few are central to our discussion; many control advanced features of the processor, see [44, Chapter 6] for more detail. Table 12.1 shows a list of pertinent registers, each has a mnemonic which we will use instead of the register number for clarity and is composed of a number of fields shown by Figure 12.1. The *BadVAddr*, *Count*, *Compare* and *EPC* registers are not very interesting in that they each consist of a single 32-bit field; *Status*, *Cause* and *Config* consist of many fields, mainly 1-bit flags. We will expand on the meaning of relevant fields in subsequent sections. For now it suffices to understand that when we refer to the *BE* field in *Config*, for example, we mean the 15-th bit of $CP0[16]$; this bit indicates the endianness mode the processor is operating in. For more detail, again see [44][Chapter 6].

There are three additional points of information not captured by the bit-field diagrams. Firstly, some fields are read/write in the sense that we can read their value and write a new value to update the register content, others are read only in the sense that we can only read their value. Secondly, in *Status* the combination of UM and $R0$ fields has the alias KSU so in some sense you can think of $KSU_1 \equiv UM$ and $KSU_0 \equiv R0$. Thirdly, just like it only supports some co-processor registers, SPIM does not support all fields in all registers. For example, in *Status* roughly only those fields related to interrupt handling are supported.

12.2.2 MIPS32 Processor Modes

In common with other processors, MIPS32 permits four **processor modes** which one can also think of as **privilege levels**. The current mode is dictated by the value of various status registers. The processor is in:

Debug Mode	when	DM field in $Debug$ is $1_{(2)}$
Kernel Mode	when	DM field in $Debug$ is $0_{(2)}$ KSU field in $Status$ is $00_{(2)}$ EXL field in $Status$ is $1_{(2)}$ ERL field in $Status$ is $1_{(2)}$
Supervisor Mode	when	DM field in $Debug$ is $0_{(2)}$ KSU field in $Status$ is $01_{(2)}$ EXL field in $Status$ is $0_{(2)}$ ERL field in $Status$ is $0_{(2)}$
User Mode	when	DM field in $Debug$ is $0_{(2)}$ KSU field in $Status$ is $10_{(2)}$ EXL field in $Status$ is $0_{(2)}$ ERL field in $Status$ is $0_{(2)}$

The kernel "owns" some region of memory which contains data and instructions that allow it to operate; we call this **kernel memory** to contrast with **user memory** which is "owned" by user processes. The basic idea is that depending on the current mode when it is executed, a given instruction will be allowed or disallowed access to a certain processor feature or region of memory. Intuitively this is a useful feature since it enables the kernel to protect resources it manages against misuse by executing processes. For example, it might be reasonable for changes to the processor configuration to be restricted to kernel mode (i.e., performed by the kernel) since if this were allowed in user mode (i.e., performed by a user process), one process could incorrectly configure the processor and perhaps cause another process to execute incorrectly.

Two questions arise from this description. Firstly, what is the consequence of an instruction executing in the wrong mode ? Above we said it could be disallowed access to some resource. In reality we would like to catch the event and have the kernel deal with it gracefully. This is achieved via the exception and interrupt mechanism described in the next section. Secondly, how does the processor enter and exit kernel mode ? If kernel mode is intended to implement some protection mechanism, what is to stop a process executing in user mode from simply changing the processor state to kernel mode and subverting that protection ? And if this change is disallowed, how can we *ever* enter kernel mode ? Again we will see that this is achieved via the exception and interrupt mechanism.

12.2.3 MIPS32 Assembly Language

At a simple level the kernel is just a special piece of software and as such we implement it using assembly language just like we might implement a program intended

to be run in user mode. The only real difference is the level and detail at which we need to interact with the resources the processor offers. To cope with this, it is useful to extend the discussion in Chapter 10 with some special-purpose assembler directives.

In particular, we need to use `.kdata` and `.ktext` which are analogies of the `.data` and `.text` directives already described in Chapter 10. Using `.data` and `.text` allowed the programmer to mark sections of data and instructions so the assembler placed them in relevant places within the object or executable file, and allowed it to prevent errors such as the use of invalid content within a particular section. The `.kdata` and `.ktext` directives work in the same way except they specify kernel data and instruction sections. Both directives accept an optional address argument that allows placement of data or instructions at a particular address. For example, we can use the directive

```
.ktext 0x80000180
```

to place code at address $80000180_{(16)}$.

A second set of new directives reduces how pedantic the assembler is when preventing access to reserved registers. Whereas normally $GPR[1]$ (or `$1`, or `$at`) is reserved for use by the assembler, by issuing the directive

```
.set noat
```

we are able to use it as normal, e.g., write to $GPR[1]$ without provoking an error. Using the directive

```
.set at
```

again makes it reserved; the assembler will again produce an error if $GPR[1]$ is written to.

12.3 Boot-Strapping

You probably know the "chicken or egg" problem: given a chick hatches from an egg laid by another chicken, how did the first egg (or chicken) come about ? There is a similar problem in the context of programs and their execution. When a computer is powered on, the memory content is essentially empty (or at least just random) so although one can reset the processor state to start executing a program as some fixed address, the program image needs to be there in the first place. The problem is, we already described the kernel as being responsible for management of loading and execution of programs: the kernel is also a program but clearly it cannot load itself ! To solve this problem, it is common for a short **boot-strap** program to be stored in non-volatile ROM. Often this forms part of very low-level configuration called the **Basic Input/Output System (BIOS)** which allows direct control of how hardware devices should behave. For example, the BIOS often includes options that control how the memory hierarchy behaves, allowing one to enable or disable cache memories and so on.

The boot-strap program itself is tasked with **booting** the computer. The idea is that when the processor is powered on, it resets into kernel mode and is fed the boot-strap program which initialises the hardware components, and invokes a dedicated **boot-loader** program which loads the kernel and starts executing it. For example, a typical boot-loader might first check the floppy disk drive (if one exists) for presence of a kernel, then move onto any hard disk drive or CDROM and so on. The exact mechanism by which it does this is beyond our scope, but clearly the program can be fairly simple since it has a very specific task. In addition, the boot-strap and boot-loader programs can typically be kernel agnostic in the sense that they do not care too much about *which* kernel they are loading, just that the executable program starts at one of the preset locations. Since the processor starts in kernel mode, the boot-strap and kernel have full access to all resources. Once the kernel is running, it takes control of the management of those resources and executes other programs in user mode.

As an aside, the slightly odd terminology used is derived from the phrase "to pull one's self up by one's boot-straps". The original source of this phrase is debatable, but the implication is the achievement of something seemingly impossible: in this case managing to get the computer into a working state from an uninitialised starting point.

12.4 Event Management

The terms **exception**, **interrupt** and **trap** are often used somewhat interchangeably, but in reality they are different and it *can* be important to see what the differences are. All basically describe an event occurring inside or outside the processor which changes the normal flow of execution in some way: usually normal processing is suspended when such an event occurs and control is passed to an event **handler** (an interrupt or exception handler) in the kernel which deals with it in some way. Roughly speaking, we can describe all three terms as follows:

Exception An exception is a condition that occurs within the processor itself. An example is an unaligned load or store, or division by zero. In these cases, something has happened that the processor cannot deal with correctly; typically this might terminate the active process, or mean that some flag is set within a status register so that the process can check whether the error occurred and continue normally.

Interrupt An interrupt is a signal from either an external hardware component that is attached to the processor (a hardware interrupt) or software component executing on it (a software interrupt, or trap), rather than the processor itself. Occurrence of an **interrupt request** from such a component indicates that normal execution should halt and some action should occur in order to manage the signalling hardware or software. Note that use of an interrupt mechanism removes the need for a **polling loop** whereby the processor would otherwise need to con-

tinuously check the hardware or software to see if it requires management (and hence waste time as a result).

Whereas an interrupt occurs because we have explicitly set things up so it will be triggered, an exception happens because of an implicit side effect within execution. Put a different way, an interrupt may occur at any time: such an event is asynchronous with respect to processor execution. Exceptions on the other hand are usually synchronous in that they occur as a direct result of a particular instruction being executed. Two final concepts that relate to both exceptions and interrupts need to be described:

Definition 48. Events are termed **precise** if when they occur and control is passed to the appropriate handler the *PC* is saved somewhere, and the pipeline content is valid in the sense that all instructions before the event are fully executed and all after the event are not fully executed. If these conditions are not satisfied, the event is termed **imprecise**.

Definition 49. Events are either **maskable** or **non-maskable**, properties that relate to whether the processor can ignore them or not. Maskable interrupts, for example, can be ignored by the processor by setting an **interrupt mask**, typically within a status register. In contrast, non-maskable interrupts always require immediate attention by the processor and have no corresponding field in the status register.

One way to categorise exceptions and interrupts is by considering an exception to be a non-maskable interrupt that just happens to be generated internally by the processor rather than externally by something else. Another way, which is adopted by MIPS32 and documented in detail within [44, Chapter 5], is a two-stage approach which relates the two concepts more or less the other way around. The basic idea is that if the processor has a mechanism to cope with exceptions, then it can use this to service interrupt requests. That is, once it detects an interrupt request the processor raises a particular type of exception to deal with the request. In order to see the complete picture of how both are managed, we look in detail at the role of the processor and the kernel.

12.4.1 Handling Interrupts

MIPS32 can distinguish between interrupts generated by up to eight different components. Two of these are reserved for software interrupts (i.e., events relating to software components), six are reserved for hardware interrupts (i.e., events relating to hardware components). Each interrupt has a number and is associated with fields in both *Cause* and *Status* as detailed by Table 12.2. Hardware interrupt number five is a somewhat special case since it is tied to the **timer interrupt** which is generated periodically by the processor itself rather than another hardware component.

The $IM0 \ldots IM7$ fields in the *Status* register represent the interrupt mask. The idea is that if, for example, $IM2$ is set to zero, then the processor is masking (or

Interrupt Type	Interrupt Number	Field in *Cause*	Field in *Status*
Software	0	*IP*0	*IM*0
	1	*IP*1	*IM*1
Hardware	0	*IP*2	*IM*2
	1	*IP*3	*IM*3
	2	*IP*4	*IM*4
	3	*IP*5	*IM*5
	4	*IP*6	*IM*6
	5	*IP*7	*IM*7

Table 12.2 A table describing MIPS32 interrupt types and their mapping into *Cause* and *Status* registers.

ignoring) hardware interrupt number zero. As such, the processor can be selective in which interrupts are dealt with in a particular context; while dealing with a critical task it might be prudent to ignore interrupts generated by the keyboard device for example since these are less important. In addition, the *IE* field in the *Status* register acts as a master interrupt enable switch: when set to one the interrupt system is enabled, when set to zero it is disabled. As such, one might include instructions at the beginning of a program such as

```
li   $t0, 0xFF01 ; move 0xFF01 into GPR[8]
mtc0 $t0, $12    ; move GPR[8] into CP0[12]
```

which load the value $FF01_{(16)}$ into $CP0[12]$, i.e., the *Status* register, to enable interrupts via the master flag and mask none of them (i.e., allow all to occur).

The behaviour of the processor when it spots an **interrupt request** from some component, when we say an interrupt is **raised**, is conceptually simple although can be tricky to implement correctly. The component in question issues an interrupt request by setting one of the $IP0 \ldots IP7$ fields in *Cause*. For software interrupts, the process can directly update the *Cause* register; for hardware interrupts the hardware component is reliant on some mechanism that sets the field for it. For example, the field could be directly connected to a pin so that an external hardware device can signal directly into the processor. An interrupt is raised when all of the following conditions are met:

1. One of the interrupt request fields in *Cause*, say *IPn*, is set to one.
2. The corresponding interrupt mask field *IMn* in *Status* is set to one, i.e., the interrupt is not being ignored.
3. The *IE* field in *Status* is set to one, i.e., interrupts have been enabled.

Once the conditions are met, the processor raises an exception which deals with the interrupt. Before it does this, however, the processor updates the *Status* and *Cause* registers to reflect the change in state. For example, it might set the *NMI* field in the *Status* register so the handler could tell the event was maskable or non-maskable.

12.4.2 Handling Exceptions

The occurrence of an exception, whether generated to manage an interrupt or a "real" exception condition such as an arithmetic error, means transfer of control to the exception handler within the kernel. Since the processor needs to know where the handler is, it must be placed at a fixed address in memory which is termed the **exception vector**. Where there is a single vector for all exception types, we term the system **single vectored**; a handler placed at a single address must work out *which* exception occurred by decoding the various status registers. In contrast, a **fully vectored** system is where dedicated addresses are used to invoke different handlers for each exception type. Trade-offs clearly exist for each choice. For example, with a fully vectored system the handler needs to do less work in decoding the exception type, but each handler is somewhat constrained in size since one handler cannot overlap into the space reserved for entry into another handler. MIPS32 implements what one might call a **partly vectored** approach. There is a single vector $80000180_{(16)}$ through which "general" exceptions are passed; a handler placed there determines the exception type by looking at the *ExcCode* field in the *Cause* register. Additionally, there are other vectors through which more "special" exceptions are passed. For example, there are dedicated vectors for dealing with non-maskable interrupts, soft reset, cache errors and so on. We concentrate on the use of a "general" exception handler from here on.

To transfer control to the exception handler, the processor must first be set into a stable and known state. Depending on the pipeline state and the need for precise exceptions the processor might first squash or flush partly executed instructions, or stall the exception mechanism until partly executed instructions are completed. Some types of kernel are **re-entrant** in the sense that, for example, they themselves can be interrupted while handling an interrupt. This is a tricky condition to deal with, so for more simple kernels which are not re-entrant, the processor might additionally disable interrupts so the kernel cannot be interrupted. The crucial next step is to switch into kernel mode: the exception needs to be managed by the kernel so to do this, the handler needs to execute in kernel mode with full access to the resources available. The exception handler can now take control; this is achieved by first setting the *ExcCode* field in the *Cause* register, and the Exception Program Counter or *EPC* register that holds the *PC* value related to the exception that occurred. For precise exceptions, *EPC* is the instruction which caused the exception (for example a load instruction which uses an unaligned address). To continue execution, *EPC* should be incremented before being used as the new *PC*. For imprecise exceptions, *EPC* is the *PC* value where execution should resume; no increment is required. Finally the processor sets *PC* to the appropriate address, in this case $PC = 80000180_{(16)}$, i.e., the "general" exception vector. Restarting execution at the new *PC* effectively invokes the exception handler which can the perform whatever tasks are required to manage the event that occurred. Once the exception handler has finished, control needs to be passed back again; typically this means restarting the process that was executing when the event occurred. The processor manages the switch from user mode into kernel mode automatically; however, a switch in

the other direction is as the result of executing the `eret` instruction which takes the content of *EPC* and copies it into *PC*, switches into user mode and then allows execution to continue.

Note that the cost of an exception (and hence an interrupt since they are managed in essentially the same way) can be described as the sum of

- the cost of entering the kernel, i.e., transferring control from user to kernel mode,
- the cost of any activity required of the kernel, e.g., dealing with an exception,
- the cost of exiting the kernel, i.e., transferring control from kernel to user mode.

Clearly it is advantageous to keep this cost as low as possible: the faster the steps are completed, the less time is wasted as overhead and the more time can be spent performing useful computation. Also note that this description resolves a problem from earlier where we wondered if a process executing in user mode was prevented from entering kernel mode freely for reasons of protection, how could we *ever* enter kernel mode ? The answer is now clear: the exception mechanism implements the switch from one to another. Either the kernel configures the processor so it is always invoked at some point (e.g., it configures the timer interrupt to be triggered every say n seconds), or it is invoked when and only when some event occurs that requires it to intervene.

12.4.3 Handling Traps

At the start of the chapter we described one of the requirements on the kernel as provision of a core set of functions which can be called by and executed on behalf of a process running in user mode. The example we gave was that of a **device driver** which is the generic name we give to operations that relate to management of a specific peripheral device. However, the concept is more general than this: imagine there is a set of **kernel functions** within the kernel which need to be called in a similar way to **user functions** housed within the process itself. The problem is, the kernel functions cannot be linked to user functions otherwise we would effectively include the entire kernel in *every* object file produced by the linker ! The typical approach to solving this general problem is the use of software interrupts as a means of implementing **system calls**, i.e., calls from a process executing in user mode into functionality within the kernel.

Since we have already explained how interrupts and exceptions work in MIPS32, the concept of a software interrupt or trap is easy to explain: essentially a process executes a dedicated instruction which raises an interrupt request that is eventually serviced as an exception by the exception handler. MIPS32 has three main classes of trap instruction:

- Execution of the `syscall` instruction is intended to represent a system call by the executing process; it results in the system call exception *Sys* (exception number 8).

- The `break` instruction is intended to cause a break-point exception which is required for single stepping through a program by a debugger; it results in the break-point exception *Bp* (exception number 9).
- There are a range of trap instructions such as `teq` and `tne` which cause the generic trap exception *Tr* (exception number 13).

We concentrate here on use of `syscall` to perform system calls. In concept, this is fairly simple. Recall that in Chapter 10 we used the `jal` instruction to implement function calls: a user mode caller executed `jal` to save the return address and then transfer control to a user mode callee function. One can consider execution of `syscall` as somewhat similar in the sense that a user mode caller executes it to transfer control to a kernel mode callee function that exists within the kernel. The open question, however, is how one passes arguments and return values between user process and kernel ? Just like the calling convention between a caller and callee user function might be standardised but is essentially a decision made by a particular programmer (or compiler), the system calling convention is a decision made by a particular kernel. That is, each kernel might have a different scheme to achieve this. A typical approach is to adopt a similar scheme to that described in Chapter 10 whereby arguments and return values are passed in predetermined registers: the user process loads the registers with values which are passed as arguments to the system call; the system call loads the registers with values which are passed back again as return values.

In fact, this approach is adopted by a limited kernel implemented by SPIM that allows simulated programs to interact with the simulation environment. For example, it allows a simulated program to print output to the terminal in which SPIM is executed. Appendix A outlines the system calling convention used by SPIM; briefly one loads the system call number into `$v0` and any arguments into `$a0`, `$a1` and `$a2` then uses the `syscall` instruction to provoke a system call trap. As an example, consider using the `print_int` system call (system call number 1) to print the integer 999 to the terminal. This can be achieved as follows:

```
l0: ...
    addi    $a0, $0, 999 # set system call argument
    addi    $v0, $0,   1 # set system call number
    syscall              # invoke system call
l1: ...
```

Instructions before the system call (denoted by the label `l0`) are executed, then we load the argument and system call number into the appropriate registers. Execution of `syscall` provokes the trap which is handled by the kernel; the integer 999 is printed to the terminal. When the kernel has finished management of the trap, control is returned and instructions after the system call (denoted by the label `l1`) are executed.

Exception Number	Exception Name	Description
0	*Int*	Interrupt.
1	*Mod*	TLB modification.
2	*TLBL*	TLB exception (load).
3	*TLBS*	TLB exception (store).
4	*AdEL*	Address exception (load or fetch).
5	*AdES*	Address exception (store).
6	*IBE*	Bus error (fetch).
7	*DBE*	Bus error (load or store).
8	*Sys*	System call.
9	*Bp*	Breakpoint exception.
10	*RI*	Reserved instruction.
11	*CpU*	Co-processor unusable.
12	*Ov*	Arithmetic overflow exception.
13	*Tr*	Trap exception.
15	*FPE*	Floating point exception.

Table 12.3 A table describing MIPS32 exception types.

```
1  .globl __start
2
3  .data
4
5  A:        .word   0
6            .word   1
7            .word   2
8            .word   3
9
10 .text
11
12 __start:  li      $t0, 0xFF01      # enable interrupts
13           mtc0    $t0, $12
14
15           la      $t0, A
16           addi    $t1, $t0,  0     # try to load from address A+0
17           lw      $t2, 0($t1)
18           addi    $t1, $t0,  4     # try to load from address A+4
19           lw      $t2, 0($t1)
20           addi    $t1, $t0,  8     # try to load from address A+8
21           lw      $t2, 0($t1)
22           addi    $t1, $t0, 12     # try to load from address A+12
23           lw      $t2, 0($t1)
24           addi    $t1, $t0, 13     # try to load from address A+13
25           lw      $t2, 0($t1)
26
27           addi    $v0, $0,  10
28           syscall                  # exit
```

Listing 12.1 An example interrupt handler (user portion).

```
.kdata

id:       .word   id0
          .word   id1
          .word   id2
          .word   id3
          .word   id4
          .word   id5
          .word   id6
          .word   id7
          .word   id8
          .word   id9
          .word   idA
          .word   idB
          .word   idC
          .word   idD
          .word   idE
          .word   idF

message:  .asciiz "unaligned access at address: "
nl:       .asciiz "\n"

save_at:  .word     0
save_v0:  .word     0
save_a0:  .word     0

.ktext 0x80000180

          .set noat
          add     $k0, $0, $at      # save GPR[1]
          sw      $k0, save_at($0)
          sw      $v0, save_v0($0) # save GPR[2]
          sw      $a0, save_a0($0) # save GPR[4]
          .set at

          mfc0    $k0, $13          # get exception id
          li      $k1, 0x3F
          and     $k0, $k0, $k1
          lw      $k0, id($k0)      # load handler address
          jr      $k0               # jump to handler address

exit_exp: mfc0    $k0, $14
          addi    $k0, $k0, 4
          mtc0    $k0, $14

exit_int: mtc0    $0,  $13          # clear cause

          li      $k0, 0xFF01       # enable interrupts
          mtc0    $k0, $12

          .set noat
          lw      $k0, save_at($0) # restore GPR[1]
          add     $at, $0, $k0
          sw      $v0, save_v0($0) # restore GPR[2]
          sw      $a0, save_a0($0) # restore GPR[4]
          .set at

          eret                      # return from handler
```

Listing 12.2 An example interrupt handler (kernel portion).

```
id0:      ...

id4:
id5:      la      $a0, message     # print message
          addi    $v0, $0, 4
          syscall
          mfc0    $a0, $8
          addi    $v0, $0, 1
          syscall
          la      $a0, nl
          addi    $v0, $0, 4
          syscall

          j       exit_exp

id6:      ...
```

Listing 12.3 An example interrupt handler (kernel portion continued).

12.4.4 An Example Exception Handler

As an example, we will consider writing a short handler to catch unaligned load and store exceptions: recall from Chapter 8 that this means the effective address used in a load or store is not a multiple of the size of data one is accessing. For example, if we access 32-bit words via the `lw` and `sw` instructions, the effective address must be a multiple of four (since a 32-bit word is four 8-bit bytes). Using the SPIM mechanism for system calls described above, we will print an error message when such an event occurs.

Although this is a specific example, the key thing to take away is the general framework which can be adapted or extended to cope with other exception types. The implementation is split into three parts. Listing 12.1 details the user portion; this represents a process executing in user mode that is used to test the handler by intentionally performing a series of aligned loads then an unaligned load to an array labelled `A`. Note that since SPIM will need to be run in bare mode without the default handler, the entry point is labelled `__start` rather than `main`.

MIPS32 permits thirty two exception types, some of which are currently unused or reserved for processor-specific behaviour. We list the first fifteen, most central types in Table 12.3; in common with co-processor register we refer to the exception name rather than number for clarity. When an exception occurs, the exception number is stored in the *ExcCode* field in the *Cause* register. The register design has the least-significant two bits set to zero. This means one can extract *ExcCode* and the least-significant eight bits of *Cause* and use them as an aligned offset into a branch table which dispatches control to a specific handler for each exception type. We use this fact to implement a fairly generic dispatcher in Listing 12.2. It is worth noting that registers *GPR*[26] and *GPR*[27], or `$k0` and `$k1`, are reserved for use by the operating system kernel: it can overwrite these registers without regard for their content whereas it must preserve others. This is an important point, and one related to the concept of preserving registers in function calls; if the kernel corrupts the

process state, then the process will execute incorrectly just like if a callee function corrupts the state of the caller, the caller will potentially execute incorrectly when control returns to it. As such, the handler starts by saving the values of `$at`, `$v0` and `$a0` which may be overwritten. It then extracts the exception type and uses the value as an offset into a branch table (which accommodates exception numbers 0 to 15 only). The branch table branches to the dedicated handler for a given type; in our case we are interested in exceptions *AdEL* and *AdES* (exception numbers 4 and 5).

Listing 12.3 lists the dedicated interrupt handler branched to by the branch table. For exception types four and five the handler prints a message followed by the address that caused the exception followed by a new line. After the dedicated handler completes, it branches back to either `exit_exp` or `exit_int` depending on whether *EPC* needs to be incremented or not as a result of a precise or imprecise interrupt. Either way, the next steps clear the *Cause* register, re-enable interrupts which had been disabled. Saved registers are then restored (note the use of the `.set noat` and `.set at` directives that allow this) before the handler uses an `eret` instruction to return to the address held in *EPC*, i.e., resume normal execution in user mode.

12.5 Memory Management

So far we have considered that processes have access to the entire **address space**, i.e., the range of addresses that are valid for use and supported by memory. As we move toward a situation where the kernel shares the processor and associated resources between many resident processes, this starts to become an unattractive assumption. We can highlight why by investigating some examples:

1. Two processes cannot both occupy the same region of memory. For example, the instructions for only one process can be located at address zero within a given address space. On the other hand, the memory model outlined in Chapter 10 stipulates that some features *should* be located at fixed addresses. For example, the stack should start at address $7FFFFFFF_{(16)}$ and grow downward. Clearly there needs to be a solution to this contradiction.
2. In the chapter introduction, we discussed the issue of memory protection using a high-level example. To briefly recap, the problem is that if several processes have access to the same address space, there is the potential for one process P to access the memory being used by another process Q. In some cases, this could be an advantage, for example P and Q can communicate with each other using the shared memory, but can also cause problems. Such problems typically take the form of accidental corruption, whereby P alters memory belonging to Q and causes it to execute incorrectly, but can also represent a security hole whereby P can read data owned by Q that it should not be able to.
3. Looking at the memory hierarchy in Chapter 8, the top levels which represent registers and memory implement an attractive trade-off: registers are fast whereas memory is large, so when we run out of registers we store (or spill) values into

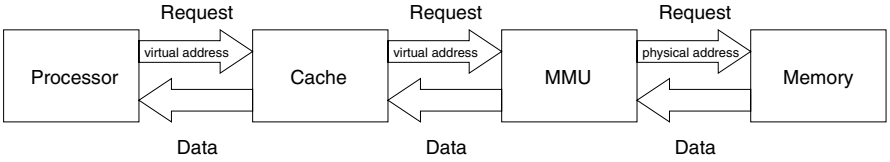
memory for a while and then load them again when we need them. At the levels below this, which represent memory and disks, we do not have the same trade-off. That is, it is obvious that when we run out of memory we could store values on disk and then load them again when we need them, we just have not discussed a mechanism to do it. Given the kernel is in control of the disks, it is reasonable to assume it should implement such a mechanism.

There are several different ways to address these problems, but we focus on a specific technique called **virtual memory**.

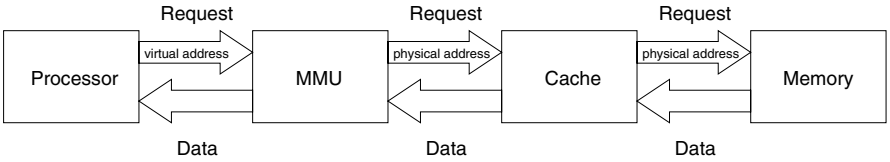
12.5.1 Basic Concept

Although a virtual memory system can offer some benefits where there is a single shared address space, we can solve all the problems above in one go by allocating each user process (and the kernel) separate address spaces. Each address space is identified with an **Address Space Number (ASN)** which could be, but does not have to be, the same as the process number. The basic idea is that instead of thinking about memory as a single large array called MEM, we split it into two concepts: an m-element **physical memory** which represents the hardware which a process can load data from and store data into, and an n-element **virtual memory** which is in some sense a process's abstract view of the address space it is allocated. To bridge the gap between the abstract virtual memory and concrete physical memory, there needs to be some mechanism managed by the kernel (and preferably supported by hardware) which translates **virtual addresses** into **physical addresses**. For example, when a process loads from virtual address x, the access is translated into a load from some physical address y.

Of course, the devil is in the detail; with virtual memory, there is quite a lot of detail and therefore plenty of devil as well. Even so, there are two key advantages from this arrangement. The first advantage is that translation mechanism permits x and y to be different on a per-process basis. To see why this is useful consider an example where we translate virtual address x to physical address p for process P and virtual address x to physical address q for process Q. The result is that both processes can access their own virtual address x (i.e., they are both fooled into thinking they have access to the full address space) but behind the scenes we are translating accesses to disjoint regions of physical memory so that both can coexist; this essentially solves the first problem from above. Even better, since each process has a separate virtual address space the second problem disappears as well: the translation process prevents one process from accessing physical memory which is used to house part of the virtual address space for another process. Essentially we get a form of memory protection for free. The second advantage is that the size of the virtual memory address space can be larger than the physical address space if we can cope with this in the translation mechanism, i.e., we can have $n > m$. This is possible because at whatever level you look at, a process is accessing a **working set** of instructions and data, i.e., those which are currently being used. We only need to



(a) MMU placed after the cache; cache operates with virtual addresses.



(b) MMU placed before the cache; cache operates with physical addresses.

Figure 12.2 A diagram showing two options for placing the MMU and the implication for how cache memory operates.

keep the working set in physical memory and can switch regions in and out behind the scenes to ensure this; this essentially solves the final problem from above.

As one might expect, the sticking point is finding an efficient way to realise the translation mechanism: given the so-called von Neumann bottleneck already noted in Chapter 8, the mechanism should not impose a performance overhead and ideally not an area overhead either. There are a few different approaches but we will examine a standard approach termed **paged virtual memory**. Conceptually, there is a convenient similarity between how cache memory and paged virtual memory operate. At a high-level, therefore, paged virtual memory can be described by the following comparison:

A cached memory system:

- divides memory into chunks called cache lines,
- operates transparently; configuration (e.g., dirty and valid bits) for each cache line managed internally and automatically by cache,
- holds a sub-set of larger address space; cache lines not held in the cache are stored in memory
- uses address translation to map addresses to sub-words within cache lines,
- uses management policies control cache line replacement, eviction and so on.

A paged virtual memory system:

- divides memory into chunks called **pages**,
- operates with support from the kernel; configuration (e.g., dirty and valid bits) for each page held in a **page table**,
- holds a sub-set of larger address space; pages not held in physical memory are stored on disk in a dedicated **swap file**,
- uses address translation to map virtual address to frame index and offset within physical memory,
- uses management policies control page replacement, eviction and so on.

The virtual memory system is often supported by the addition of a dedicated **Memory Management Unit (MMU)** which works alongside the processor so when accesses to memory are made, the accesses pass through and are partly serviced by the MMU. The MMU, and the decision to use virtual memory at all, is closely integrated with the processor ISA and micro-architectural design. For example, exactly where the MMU is placed highlights implications of the difference between virtual and physical addresses. Previously, our processor and associated components dealt exclusively with physical addresses. Now that a process will access memory using virtual addresses that are translated into physical addresses by the MMU, changes need to be made: any component which previously stored or operated on physical addresses now potentially needs to cope with virtual addresses instead. For example, the *PC* associated with each process will now be a virtual address and instruction fetches will thus be to virtual addresses. Likewise, any cache memories or branch prediction hardware will potentially need to change: in the case of caches, this depends on where the MMU is placed. Figure 12.2 shows two options, one where the MMU is placed before the cache and one after. In the first case, the cache operates with virtual addresses. On one hand this seems reasonable, but a potential problem is that if the cache tags are based on virtual addresses and there are many virtual address spaces, there is a danger that the cache will incorrectly identify two identical addresses from different address spaces as the same: this effectively means the wrong cache line will be resident. To cope with this the cache tags could be extended to include an ASN, or the cache flushed each time a new process (and hence address space) becomes active. The second case places the MMU before the cache so that it operates on physical addresses so that things work roughly in the same way as before.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char* argv[] )
{
  printf( "page size is %ld bytes\n", sysconf( _SC_PAGESIZE ) );
}
```

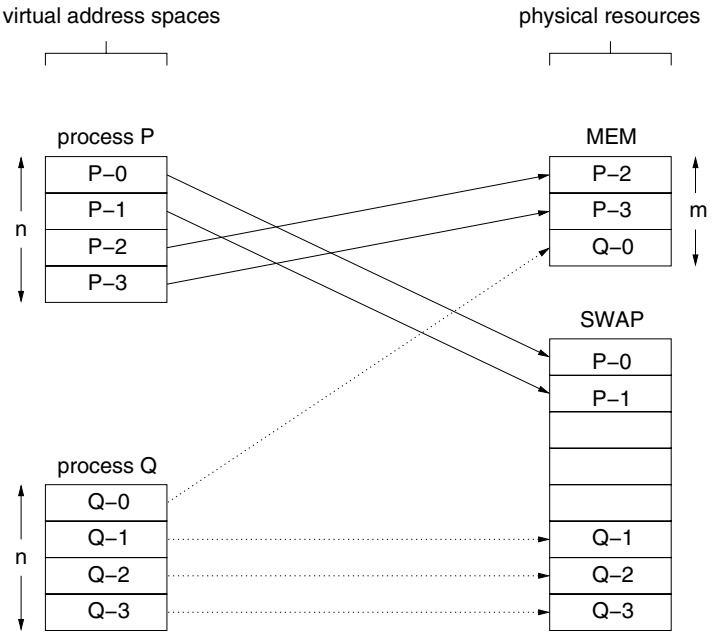
Listing 12.4 A short C program to test the page size used by the host operating system.

12.5.2 Pages and Frames

The basic idea is to divide physical and virtual address spaces into fixed size chunks. The chunks of physical address space are called **frames**, there are VM_{frames} of them and each one is $VM_{framesize}$ in size so that the physical address space is $m = VM_{frames} \cdot VM_{framesize}$ bytes. The chunks of virtual address space are called **pages**, there are VM_{pages} of them and each one is $VM_{pagesize}$ in size so that each virtual address space is $n = VM_{pages} \cdot VM_{pagesize}$ bytes. The size of frames and pages are fixed and the same, i.e., $VM_{framesize} = VM_{pagesize}$, but can be dictated by the kernel. One can query the page size, for example, using a program such as the one outlined in Listing 12.4. A representative size would be 4096 bytes, and one would expect the size always to be a power-of-two in order that other aspects of the system are made simpler.

Each user process (and the kernel) has a separate virtual address space identified by an ASN. At a given point in time, frames in physical memory will be populated by pages from these address spaces as detailed by the example in Figure 12.3. Imagine that $VM_{framesize} = VM_{pagesize} = 4096$ bytes and that physical memory can, somewhat oddly, hold only $VM_{frames} = 3$ frames. The physical address space is thus $m = VM_{frames} \cdot VM_{framesize} = 3 \cdot 4096 = 12288$ bytes so valid physical addresses are in the range $0 \ldots 12287$. Two processes P and Q are resident, each of which has a virtual address space that consists of $VM_{pages} = 4$ pages. Each virtual address space is thus $n = VM_{pages} \cdot VM_{pagesize} = 4 \cdot 4096 = 16384$ bytes so valid virtual addresses are in the range $0 \ldots 16383$. For clarity the i-th page in the virtual address space for P or Q is labelled $P-i$ or $Q-i$, for example in this case process P has four pages labelled $P-0$, $P-1$, $P-2$ and $P-3$.

Frames in the physical memory MEM are currently populated with the three pages $P-2$, $P-3$ and $Q-0$. The remaining five pages (i.e., pages $P-0$, $P-1$, $Q-1$, $Q-2$ and $Q-3$) are held on disk in the swap file $SWAP$. Based on this state, it is already important to re-enforce two facts. Firstly, frame number i in physical memory does not necessarily hold page number i from a particular virtual address space; the translation mechanism maps pages to arbitrary frames. Secondly, frame numbers i and j do not necessarily hold pages from the same address space; physical memory can be populated with pages with different owners but the translation mechanism gives the illusion that each has dedicated access.



$PT_{i=0}$

	$PT_i[j]_{location}$	$PT_i[j]_{swap}$	$PT_i[j]_{frame}$	$PT_i[j]_{dirty}$
$j=0$	$SWAP$	0		
$j=1$	$SWAP$	1		
$j=2$	MEM		0	0
$j=3$	MEM		1	0

$PT_{i=1}$

	$PT_i[j]_{location}$	$PT_i[j]_{swap}$	$PT_i[j]_{frame}$	$PT_i[j]_{dirty}$
$j=0$	MEM		2	1
$j=1$	$SWAP$	5		
$j=2$	$SWAP$	6		
$j=3$	$SWAP$	7		

Figure 12.3 A diagram showing pages from two processes variously resident in physical memory and swap, and the page tables which describe this example state.

To keep track of the virtual memory state, the kernel maintains an internal data structure for each ASN which describes the associated virtual address space. The organisation of this data structure is crucial, but in order to focus on the fundamental idea we take a (very) simple approach:

Definition 50. For each virtual address space there is a **page table** where we write PT_i to describe the page table related to the ASN i. Each PT_i has VM_{pages} entries in

it, one for each page in the associated virtual address space. The j-th entry in such a page table, which we write $PT_i[j]$, has the following fields:

1. $PT_i[j]_{location}$ which determines whether page j resides in physical memory or the swap file.
2. If page j is resident in the swap file:
   - $PT_i[j]_{swap}$ holds an offset into the swap file where the page content is stored.
3. If page j is resident in physical memory:
   - $PT_i[j]_{frame}$ holds the frame number in physical memory in which the page is resident.
   - $PT_i[j]_{dirty}$ determines if the page has been altered or not (i.e., if the versions in memory and on disk are different).

Although this simple scheme will suffice for our description, several issues need to be at least mentioned if not explored further. Firstly, note that it can be attractive to split this description into separate parts; often one might find a dedicated **frame table** to manage frames in physical memory or a dedicated **disk map** to manage pages in the swap file. Secondly, note that the data structure used to realise the information held in a page table is crucial: many techniques for efficient page table data structures exist (e.g., so-called inverted or multi-level page tables) although we ignore them here. Thirdly, note that the kernel might reserve some of the frames in physical memory as special, and lock them so they cannot be used to hold pages from address spaces associated with user mode processes. The reason for this is simple: if the kernel itself does not exist in physical memory, it will be hard for it to manage the various data structures ! To avoid this somewhat circular problem, we simplify matters and assume the address space associated with kernel memory is *always* resident in physical memory.

Based on this description, the bottom half of Figure 12.3 enhances the top half by describing two page tables PT_0 and PT_1, which relate to ASNs 0 and 1 assigned to processes P and Q and allow us to interpret the state of the virtual memory system. A **demand paging** system [25] would assume that when a process is initially allocated a page table no pages are resident in physical memory: all pages are transferred into physical memory or the swap file on demand, i.e., when their content is accessed. In this case, however, we are past this initial stage since frames in physical memory are populated with pages from P and Q. In particular, looking at the PT_0 we can see that page 1 in this address space (i.e., page $P-1$) is resident in the swap file at offset 1. On the other hand, looking at PT_1 we can see that page 0 in this address space (i.e., page $Q-0$) is resident within frame 2 of physical memory and has been altered (i.e., any version of the page within the swap file is out of date).

12.5.3 Address Translation and Memory Access

Given suitable page tables that are configured by the kernel to describe the state of the virtual memory system, the next step is to utilise them so memory accesses can be satisfied: this is where the MMU starts to play a role. The MMU needs access to the page table for the process currently active (i.e., the process currently executing on the processor) which would have been configured by the kernel. Conceptually, and in MIPS32 at least, one can imagine the kernel setting the co-processor registers to reference the current ASN and include a pointer to the associated page table data structure. An important fact to re-enforce is that the MMU operates as autonomously as possible. We do not want every memory access by a process to require attention by the kernel since this would impose a massive performance overhead. Rather, the MMU is able to automatically satisfy memory accesses for virtual addresses whose corresponding page is in physical memory; the kernel is only invoked when there needs to be some management or re-configuration of the virtual memory system.

Put simply, every use of a virtual address VA must be translated into a physical address PA which can be serviced by the physical memory MEM. Examples where accesses using VA are performed are easy to imagine: our PC must now hold the virtual address of the next instruction to fetch from memory, and execution of a load or store instruction will require a virtual effective address. The translation process is similar to that we described while looking at cache memories: we use VA to calculate two components which are used to access the active page table:

$$\begin{aligned} VA_{offset} &= VA \bmod VM_{pagesize} \\ VA_{index} &= \lfloor VA / VM_{pagesize} \rfloor \end{aligned}$$

The value VA_{index} gives the page number for this virtual address, it identifies the page sized chunk of our virtual address space the virtual address lies within; the value VA_{offset} gives the offset within that page. Since the kernel would select $VM_{pagesize}$ to be a power-of-two, computing the values VA_{index} and VA_{offset} is not costly: given a 32-bit address VA, if we select $VM_{pagesize} = 4096$, then it amounts to taking the twelve least-significant bits as VA_{offset} and the remaining twenty bits as VA_{index}. Returning to the example in Figure 12.3, one could imagine process number 0 (i.e., process P) performing a load from virtual address

$$VA = 00003008_{(16)} = 12296_{(10)}.$$

This is reasonable: the virtual address $12296_{(10)}$ is within the range of valid virtual addresses even though it is outside the range of valid physical addresses. Extracting the correct sections of VA as follows

$$\underbrace{00003}_{VA_{index}}\underbrace{008}_{VA_{offset}}{}_{(16)},$$

we find that

$$\begin{aligned} VA_{offset} &= \quad 008_{(16)} = 8_{(10)} \\ VA_{index} &= 00003_{(16)} = 3_{(10)} \end{aligned}$$

or more simply that this virtual address is in page number 3 (i.e., page $P-3$) at offset 8 of this virtual address space.

The MMU can check if the page is resident in physical memory by inspecting

$$PT_i[VA_{index}]_{location}.$$

That is, it looks in the page table for the active process and retrieves the field which tells us whether page VA_{index} is held in physical memory or the swap file. As is the case detailed in Figure 12.3, imagine the page is held in physical memory. In this case, the easier of the two, the MMU can translate the VA into a physical address PA as follows:

$$PA = PT_i[VA_{index}]_{frame} \cdot VM_{framesize} + VA_{offset}.$$

Step-by-step, the idea is that the MMU looks in the page table for the active process and retrieves the frame number for page VA_{index} which is scaled to give a base address, and added to VA_{offset} to give the physical address. That is, it calculates

$$\begin{aligned} PA &= PT_i[VA_{index}]_{frame} \cdot VM_{framesize} + VA_{offset} \\ &= PT_0[3]_{frame} \cdot VM_{framesize} + 8 \\ &= 1 \cdot 4096 + 8 \\ &= 4104_{(10)} \end{aligned}$$

The resulting address PA is within the range of valid physical addresses and can thus be used as the effective address passed to the physical memory in order to perform the load we started off with. Note that in this case the MMU has dealt with the load without any intervention of the kernel: there is essentially no performance overhead (bar the cost of actual translation). Where the memory access is a store rather than a load, the access mechanism works much the same way: if the page is in physical memory, we can simply translate the address and perform the store. However, one minor addition is required in order to update the value of $PT_i[VA_{index}]_{dirty}$. This marks the page in the page table as dirty which means that the version held in physical memory is noted as differing from the version held in the swap file: essentially we know that the page content has been updated as a result of the store instruction being executed.

The more difficult case is where page VA_{index} is not held in physical memory: the value we want to load is currently resident in the swap file so clearly we cannot perform the load. Imagine that process number 0 (i.e., process P) performs another load, this time from virtual address

$$VA = 00000008_{(16)} = 8_{(10)}.$$

This address gives $VA_{index} = 0$ and $VA_{offset} = 8$, i.e., we are trying to load from page number 0 (i.e., page $P-0$) that is currently held in the swap file. This sort of situation is termed a **page fault** and results in an exception being raised that invokes the kernel in order to resolve matters so the load can eventually be performed. You can think of this as being similar in concept to the exceptions caused by unaligned loads (or stores) we examined previously: in both cases a load (or store) instruction causes some problem with respect to the memory system that needs to be resolved by the kernel.

12.5.4 Page Eviction and Replacement

When a cache memory discovers that it does not hold the cache line associated to some address, it might first evict an old cache line to free some space and then load the required cache line from memory. One might say the old cache line is replaced by the new one; once this occurs, the cache can perform the access required of it. The kernel performs a similar task with pages. As above, imagine that process number 0 (i.e., process P) performs a load from virtual address

$$VA = 00000008_{(16)} = 8_{(10)}$$

which gives $VA_{index} = 0$ and $VA_{offset} = 8$. To perform the load, page 0 (i.e., page $P-0$) must be held in physical memory so the kernel must first evict another page to free space and then load page 0 from the swap file. Once the page is loaded into physical memory and the page table updated to reflect the new state, the kernel can return control to the process which caused the page fault thereby restarting the original memory access: the second time around this will succeed because page 0 is now in physical memory. If one believes that access to the swap file on disk is possible, this should seem a fairly simple task for the kernel. However, there are at least two issues that need to be mentioned which can somewhat complicate matters.

Firstly, how does the kernel select a page for eviction ? That is, if there are no empty pages (in which case the kernel would just select one of them to load the new page into) which page of those resident should be moved from physical memory into the swap file ? Like a cache, the kernel needs to employ a replacement policy so those pages used most are likely to be retained in physical memory. Although the policy is analogous to that required in the context of caches, since the cost of accessing memory and disk is higher than the cache and memory the penalty for an ineffective policy is higher as well. The particular case where a page is repeatedly swapped into physical memory then swapped out again is often called **thrashing**. The prolonged spells of disk activity that result mean you can often hear this happening as well as noticing the performance degradation: the system seems to "freeze" as the kernel spends most of the time swapping pages in and out of physical memory and no time allowing processes to execute. When we examined caches, use of Least Recently Used (LRU) policy was described as an expensive option despite

providing an effective way to identify cache lines to evict. As a result of the higher penalties involved here, the overhead of operating LRU to identify pages to evict is typically worth it.

Secondly, what actually happens when a page is evicted ? Like a cache, if the page resident in physical memory that we want to evict is dirty (i.e., the page content has been updated), we first need to copy back the content to the swap file. More simply, if we want to evict a dirty page, we first save the content into the swap file whereas if the page is not dirty we can simply discard it. In the context of our running example from above process number 0 (i.e., process P) needs page 0 (i.e., page $P-0$) in physical memory. If the kernel chooses to evict the page in frame 0 (i.e., page $P-2$), then it can simply overwrite the frame with new page content since the old page content is non-dirty. However, if the kernel chooses to evict the page in frame 2 (i.e., page $Q-0$), then it first needs to save the old page content into the swap file, since it is marked as dirty, and only then overwrite the frame with new page content. This difference in behaviour might influence the page replacement policy in the sense that if we select dirty pages for eviction, the cost is higher than if we select non-dirty pages. Specifically, the former case implies two accesses to the swap file (one page store and one load) whereas the latter implies one access (one page load).

12.5.5 Translation Look-aside Buffer (TLB)

So far we have painted a very positive picture of virtual memory: as long as the kernel can manage the page table configuration, the initial problems we outlined are more or less solved. That is, by the kernel and MMU implementing the translation mechanism behind the scenes and swapping pages in and out of physical memory, each resident process has access to a large protected address space. However, there is a nasty issue lurking behind this picture. At the moment, each access to memory performed by a process actually requires two accesses to memory, one to inspect the page table and one to perform the actual access. Clearly this is far from ideal: even though the MMU acts autonomously so as to minimise the kernel activity to those occasions where some management or re-configuration of the virtual memory system is required, the price of solving our problems has been a 2-fold increase in the cost of every memory access. Since this includes instruction fetches, we have essentially halved the performance of the processor !

Understandably, solutions to this issue are fundamental if virtual memory is to be feasible for use. Fortunately we have already examined techniques that can provide such a solution: the basic idea is to hold page table entries in what amounts to a fancy cache memory called a **Translation Look-aside Buffer (TLB)**. Although there are ways to optimise the performance of the resulting system, in a basic sense the TLB is just a fully-associative cache which maps page numbers to frame numbers. Essentially we try to use the TLB to eliminate memory accesses that inspect the page table in our translation mechanism, i.e., we want to have a situation where

$$PA = \underbrace{PT_i[VA_{index}]_{frame}}_{\text{retrieved from TLB}} \cdot VM_{framesize} + VA_{offset}.$$

That is, we calculate VA_{index} and send this to the TLB which searches the entries and hopefully returns us the base address of the frame which holds that page. Clearly the TLB can only hold a sub-set of entries from a given page table. For some virtual address VA, if there is an entry for VA_{index} in the TLB a **TLB-hit** occurs and we can quickly form the physical address PA without a second memory access. If there is no entry for VA_{index} in the TLB, a **TLB-miss** occurs and we are forced to access the page table in memory to form PA. The principles of temporal and spatial locality hopefully mean that more TLB-hits occur than TLB-misses and hence on average the second accesses required to load a page table entry from memory are required infrequently.

Since the TLB is a cache, in the case of a TLB-miss one would expect the page table entry retrieved from memory to replace one currently resident in the TLB. This can be managed automatically in hardware like a normal cache, or provoke an interrupt which allows the kernel to dictate the replacement policy.

12.6 Process Management

Informally we have already said that a process is a run-time entity in the sense that a process **context** captures one executing instance of a program: the program is a static entity, the process is an active or dynamic entity. In the most general sense, we would like to define a process as the basic unit managed by an OS that supports **multitasking**. The basic idea is that the kernel permits many processes to be **resident** in memory at the same time, keeps track of their contexts and swapping between them to allow sharing of the processor and hence concurrent execution: if virtual memory provides an abstract view of memory, one could think of multitasking as providing each process a virtual processor which is an abstraction of the real processor.

The concept of a **process** versus that of a **thread** both have associated subtleties that we need to be careful about. As such, we will adopt a (fairly) standard way to categorise the two:

Definition 51. Processes:

- have independent goals; each is executed to perform a different task,
- have separate processor context (e.g., different *PC* and *GPR* state),
- have separate address space,
- interact with each other using relatively expensive mechanisms (e.g., system calls).

Definition 52. Threads:

- have a common goal; each is executed as part of a thread team which cooperate to perform a single task,

- have a somewhat separate processor context (certainly different *PC*, probably shared *GPR* state),
- have a shared address space,
- interact with each other using relatively inexpensive mechanisms (e.g., via controlled access to their shared address space).

Again informally, one might describe a thread as a sub-process or lightweight process. Some kernels manage threads naively alongside processes, others do not recognise them and force threads to be supported by a user mode library. Here we are interested mainly in the mechanism that allows the processor to be shared, so we take the simplifying approach of assuming the two things are essentially the same and exclusively discuss processes.

12.6.1 Storing and Switching Process Context

For each process the kernel allocates a **Process Identifier (PID)**, which is simply an integer name for the process, and maintains a **Process Control Block (PCB)** that acts as an area where the process context can be stored:

Definition 53. The PCB for the i-th process, which we denote $PCB[i]$, has (at least) the following fields:

1. $PCB[i]_{PC}$, the saved program counter for the process which represents the address where execution should resume.
2. $PCB[i]_{GPR}$, the saved general-purpose register file content; *all* registers in *GPR* are saved here, we use $PCB[i]_{GPR[j]}$ to refer to the j-th such register where appropriate.

Of course, each PCB might include many more details that relate to how the kernel controls each process. For example, one might include a **priority level** whose use we will see later, and pointers to any virtual memory configuration (e.g., the page table) for the process.

With one processor, clearly only one process can actually be executing at a time: the context of that **active process** is held and used by the processor, the context of other **non-active processes** are stored in their respective PCB entries. In order for the kernel to share the processor between the processes, there needs to be a mechanism to swap the active executing process for another; this is called a **context switch**.

Imagine there are two resident processes called P and Q which have PIDs 0 and 1 respectively, and that P is the active process: the context of P is held by the processor since P is currently executing, the context of Q is held in $PCB[1]$. At some point, the kernel steps in and decides that Q should execute instead: to do this, the context associated with P needs to be saved and then replaced by the context associated with Q. More simply, the kernel saves the context held by the processor into $PCB[0]$ then

loads the saved context from $PCB[1]$. Since this replaces the value of PC, when the kernel returns control to user mode process Q will start to execute; $PCB[1]_{PC}$ has become the new PC so this is where execution recommences. Now imagine at some point the kernel again steps in and decides Q has executed for long enough and that P should have another chance: to do this, the kernel saves the context held by the processor into $PCB[1]$ then loads the saved context from $PCB[0]$. When the kernel returns control to user mode process P will resume execution; $PCB[0]_{PC}$, where execution of P was halted the first time we switched context, has become the new PC so this is where execution recommences.

A context switch is conceptually simple but deceptively costly. As in the case of handling an exception, said cost can be described as the sum of transfer into kernel mode from user mode and vice versa, plus activity performed by the kernel. Focusing on the kernel activity, to switch from the process with PID 0 to the process with PID 1 we simply save the PC into $PCB[0]_{PC}$ and each $GPR[j]$ into $PCB[0]_{GPR[j]}$, then we load PC from $PCB[1]_{PC}$ and each $GPR[j]$ from $PCB[1]_{GPR[j]}$. Even with this simple scheme where the context of a process is described by just the program counter and a register file of thirty two registers, we need a total of sixty six memory accesses ! This is a lower bound on cost: we did not for example include the cost of the kernel deciding on which process it wants to execute or updates to state such as the virtual memory configuration. Keeping in mind that context switching is pure overhead with respect to useful computation, this is a high price and one which particular architectures attempt to solve with the assistance of features in the processor.

12.6.2 Process Scheduling

Although we have established a mechanism to store and switch process context to and from the processor and PCB table, there are three questions to answer that fall roughly under the umbrella of **process scheduling**. The goal is to have a **scheduler** (a particular algorithm or mechanism within the kernel) perform context switches in order to manage access to the processor according to some policy. Firstly we need to understand what states a process can be in. For example, how do we know some process Q is a candidate to be executed and more generally, how does the kernel keep track of this information ? Secondly, given the scheduling mechanism needs to be executed in some way so it can perform the context switch, what options exist for provoking this execution ? Thirdly, given some process P will be active, *which* process (i.e., which Q) should the kernel select to replace it ?

12.6.2.1 Process States and Queues

The starting point is the definition of numerous **process states** which describe what phase of execution a process is currently in. The specific states and their level of

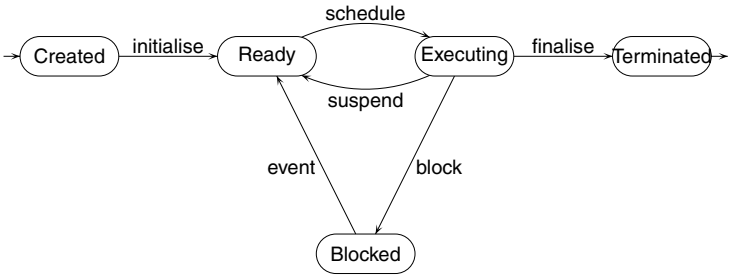


Figure 12.4 A 5-state description of a process during execution.

detail depend on the kernel; in the most basic sense one can imagine a 2-state model whereby a process could either be executing or not executing. The idea would be for the kernel to keep track of which state each process is in by maintaining a **process queue** (or list) for each possibility and moving processes between queues as their state changes. In the 2-state model this is very simple since the executing queue can only ever contain one process (if there is one processor) and the not executing queue contains all other processes.

More generally, we can formulate the 5-state description in Figure 12.4 where the meaning and purpose of each state, and the transitions between them, is as follows:

Created When a process is initially created, it is placed in the created state. This represents a "foetal" process in the sense that it has been created but not yet initialised, and is hence not yet ready for execution. Once the kernel has initialised the process (e.g., allocated structures to keep track of it, initialised context such as the instructions or stack), it is moved into the ready state and thereafter becomes a candidate for execution.

Ready From the point of view of scheduling in the sense of the question above, the ready state (and associated ready queue) is perhaps most important. Essentially, all processes in the ready state are candidates for selection by the kernel to replace the active process via a context switch; a process on the ready queue is ready for execution. The policy that selects a process from the ready queue and schedules it for execution is discussed later.

Executing A process in the executing state is active: the associated context is being used by the processor to execute instructions. During execution, one of two events can cause the active process to be descheduled (i.e., switched for some other process). Firstly, the kernel might step in and suspend the active process because it wants to (or needs to) allow another process to execute. This sets the process state to ready, and hence moves it into the ready queue. Secondly, an operation performed by the process might cause it to become blocked; for example, it could try to access a resource which is not currently free. This sets the process state to blocked, and hence moves it into the blocked queue. One slight subtlety is that at least one process *must* be available for execution otherwise

it is unclear what the processor will execute. To cope with this, many kernels maintain a somewhat privileged special process called the **idle process**: this is simply a process which causes the process to do nothing if there are no useful processes to execute.

Blocked A process which is blocked is essentially waiting for some event to happen before it can continue execution; in some sense it *can* execute but cannot do anything useful bar wait. Common reasons for a process to become blocked include waiting for some I/O operation (e.g., a slow disk access, or user input) or some resource to become available. Once the event occurs (e.g., the I/O operation completes or the resource becomes free), the process can return to the ready state since it is then ready for execution again.

Terminated During execution, a process can be terminated. Sometimes this relates to natural completion (i.e., the process exits gracefully), but sometimes can be due to action by the kernel (e.g., the kernel terminates a process that causes some exception). Either way, the idea is that the process lifetime is over and the associated context should be removed from any structure the kernel many have used to keep track of it. A process which has been terminated but whose context has not yet been removed from the kernel is termed a **zombie process**: it has been terminated but sort of still lives on !

12.6.2.2 Creating and Terminating Processes

One question that arises from the previous discussion of process states is *how* processes are created and terminated, i.e., how processes enter the "created" state initially and move into the "terminated" state at some point during execution. Both operations must be provided by the kernel as system calls: a process must be able to ask the kernel to create a new process or terminate itself. To illustrate how such system calls *can* work (a given kernel might take different approaches), we use the example of a UNIX-style kernel.

- A process can terminate itself by using the `exit` system call; the prime reason to do so would be normal completion of execution. It could also be the case that an external event triggers termination of a process, for example if the process causes an unrecoverable exception the kernel might opt to terminate it.
- A process can create a new process by using the `fork` or `exec` system calls: the original process is termed the **parent process**, the new process is the **child process**. Since the parent and child are separate processes, they are allocated separate address spaces. By using `fork`, the parent memory content is copied into the child context: the two processes execute the *same* program image. Using `exec` changes this behaviour by subsequently patching the child context with a new program image. Intuitively this can be more useful since it allows one program to initiate the execution of a totally different program. Given this model, UNIX traditionally has a root process called `init` which is the ancestor of every other process in the system: every other process has been executed as the result of a series of `fork` or `exec` started by `init`.

- The parent can use the `wait` system call to suspend itself until the child has finished executing (i.e., is terminated), otherwise the parent and child processes can execute independently. If the parent does not wait for the child and the child terminates, this is one example of where a **zombie process** occurs: the child still has a PCB entry even though it is not an actively executing process. In some cases the system enforces the convention that when the parent process is terminated, all children are terminated automatically. Where this is not the case and the parent terminates before the child, the child becomes an **orphan process**.

12.6.2.3 Invoking the Scheduler

In reality, the kernel clearly cannot just "step in" as we described above: it must be invoked by some mechanism:

Definition 54. The concept of **multiprogramming** is where the kernel waits until the active process is blocked (e.g., waiting for a read from disk to complete) and then switches it for another process. The problem with this approach is neither the kernel nor the processes have any control of when context switches occur; this makes use of multiprogramming difficult in interactive systems.

Definition 55. The concept of **cooperative multitasking** places control in the hands of the processes themselves: a context switch occurs when the active process *offers* to be switched for another process; the active process is said to **yield** control of the processor. Although this can be effective when the processes behave properly, the kernel need to deal with the issue of uncooperative processes: if the active process *never* yields, then other processes are starved of execution time.

Definition 56. The concept of **preemptive multitasking** places control in the hands of the kernel: a context switch occurs when the active process is deemed, by the kernel, *due* to be switched for another process.

Although there are advantages and disadvantages to all three approaches, an obvious advantage of preemptive multitasking is the ease by which it can be realised. More specifically, it is quite easy to construct a system which decides when a context switch is due. The easiest way to achieve this is via the concept of **time sharing** or **time slicing**. The idea is that the kernel controls a timer which acts like an alarm clock: once reset the timer waits for a period of time (a so-called **time slice** or **time quantum**) and then sounds an alarm, which in this case means it triggers an interrupt that in turn invokes the kernel. In MIPS32, this behaviour is controlled by the *Count* and *Compare* registers (i.e., registers $CP0[9]$ and $CP0[11]$). Roughly speaking, the *Count* register is incremented by the processor after completion of every cycle and when $Count = Compare$, a timer interrupt is triggered. Thus, by setting $Count = 0$ and *Compare* equal to the time quantum, e.g., $Compare = 1000$, one can force an interrupt to occur 1000 cycles into the future. In turn, this allows the kernel to force a situation where the process scheduling mechanism is invoked every 1000 cycles: a process can execute for 1000 cycles after which the kernel "steps in" and performs a context switch so another process can execute.

12.6.2.4 Scheduling Policies

Finally we are in a position to discuss how the scheduler, which we assume has been invoked by some mechanism, selects a process from the ready queue in order to perform a context switch. That is, how the scheduler applies a policy which would typically attempt to be

- fair, i.e., it does not indefinitely prevent a (valid) process from executing even though it may favour some over others,
- efficient, i.e., the scheduling mechanism does not take a prohibitively long time to execute,
- robust, i.e., it cannot perform some action which causes a process (or the kernel) to malfunction.

Often, the scheduling policy is split into three levels which are applied with differing frequencies and have differing goals:

Long-term The long-term scheduler deals with the high-level or "big picture" issues; it is invoked infrequently and tasked with deciding which processes should inhabit the ready queue. The idea is that the long-term scheduler takes on the role of load balancing: it might try to maintain a mix of I/O bound (i.e., those that perform mainly I/O) and computationally bound processes for example, in order to give the best overall system performance.

Mid-term The mid-term scheduler is typically tasked with swapping processes into and out of physical memory based on their current state (or predicted future behaviour).

Short-term The short-term scheduler deals with the low-level detail of selecting a process from the read queue and performing a context switch so that it can execute. Since the short-term scheduler is invoked frequently (e.g., after each time slice), efficiency of this operation is important.

Even within these types, one might expect different algorithms to be deployed in different scenarios in order to cope with different demands. In short, a "one size fits all" approach is seldom appropriate. Investigation of these algorithms at a deep level is a little beyond the scope of this book; to give a flavor, some very simple short-term approaches to scheduling are overviewed below:

Round Robin The concept of a "round robin" allocation is more general than this context; it means giving an equal share of some resource, in turn, to some recipients. Here, the resource is execution time and processes are the recipients: a round robin scheduler assigns a fixed size time slice to all processes and iterates through them in turn. Each time it is invoked, it selects the next process and schedules this for execution. This approach is simple and so can be inexpensive to implement and execute; it is fair in the sense that every process is guaranteed to be executed at some point.

Priority-based One of the disadvantages of round robin scheduling is that it ignores priorities that might be assigned to processes. For example, processes that

deal with interaction with the user could be said to be higher priority: if they are not scheduled very soon after a user clicks the mouse or presses a key for example, the user will notice the delay which will degrade their experience.
By assigning each process a priority, e.g., adding it to the PCB entry maintained by the kernel, one can imagine a scheduling scheme that alters round robin by ordering the iteration by priority. That is, instead of iterating through the processes in turn, it orders them so that the next process is more likely to be one of those with a high priority than one with a low priority. Of course the problem with this approach is fairness: one now cannot make the claim that every process will be executed at some point since a low priority process could be starved of execution time by higher-priority processes.

Resource-based If one could estimate the amount of resources that a process needs to execute, e.g., the length of time it is expected to execute, one might make scheduling decisions based on this. For example, one might use the so-called Shortest Job First (SJF) or Shortest Remaining Time (SRT) strategies whereby a process which is expected to take the least time to execute is scheduled before others. Intuitively this should make some sense; it roughly operates like the "express" lane at a supermarket checkout. The idea is to maximise throughput in terms of the number of processes that complete in a given time period. The disadvantages are that again fairness is hard to guarantee, and also that obtaining accurate information about execution time (or any resource requirement for that matter) can often be difficult.

12.6.3 An Example Scheduler

In an attempt to demonstrate the main concepts of process management, we will write a "process switcher" that one can think of as a (very) limited preemptive multitasking scheduler. The switcher is invoked by the timer interrupt and switches between two hard-coded processes (i.e., there is no `fork` system call), one which prints the character 'A' and one which prints the character 'B'. There is a nice historical motivation to use of this particular example. Moody [45, Page 36] recounts the words of Linus Torvalds, creator of the Linux kernel, in describing his efforts on the `comp.os.minix` newsgroup:

> *I was testing the task-switching capabilities [of the Intel 80386 processor], so what I did was I just made two processes and made them write to the screen and had a timer that switched tasks. One process wrote 'A', the other wrote 'B', and I saw "AAAABBBB" and so on.*

Like the example exception handler, this is a specific example but again the key thing to take away is the general framework which can be adapted or extended to cope with a more complete or flexible approach: just like Linux, with a few decades work by hundreds of committed programmers you too could transform the example into a real OS !

```
1  .globl __start
2
3  .data
4
5  .text
6
7  __start:  li      $t0, 0xFF01      # enable interrupts
8            mtc0    $t0, $12
9
10           li      $t0, 4           # set timer limit
11           mtc0    $t0, $11
12           li      $t0, 0           # set timer count
13           mtc0    $t0, $9
14
15 idle:     nop                      # perform idle loop
16           nop
17           nop
18           nop
19           j       idle
20
21 proc_0:   li      $a0, 0x41        # process 0 (prints A)
22           li      $v0, 11
23 loop_0:   syscall
24           j       loop_0
25
26 proc_1:   li      $a0, 0x42        # process 1 (prints B)
27           li      $v0, 11
28 loop_1:   syscall
29           j       loop_1
```

Listing 12.5 An example process scheduler (user portion).

Implementation of the process switcher is again split into three parts. Listing 12.5 details the user code which includes the two processes. Note again that since SPIM will need to be run in bare mode (and with the `-mapped_io` option) without the default handler, the entry point is labelled `__start` rather than `main`. The code starts by enabling all interrupts and setting the *Count* and *Compare* registers so as to trigger a timer interrupt after a few cycles. It then drops into an infinite loop labelled `idle` which consists of `nop` instructions and in some sense relates to the idle process described previously. With no action by the kernel, we would expect execution to never exit from this loop. However, our scheduled timer interrupt will cause the process switcher to execute two hard-coded processes marked `proc_0` and `proc_1` which loop and use the SPIM mechanism for system calls to print 'A' and 'B' characters to the terminal. Note that we have intentionally initialised the arguments to the system call outside the loop. This acts as a rough test that the scheduler is context switching correctly: if it does not correctly save and restore the `$a0` and `$v0` registers for each process, they will not print the right thing.

Listing 12.6 presents a similar generic dispatcher as the one used in the exception handler example, i.e., Listing 12.2. It performs the same sequence of saving registers used by the kernel, extracting and using the exception number to invoke a dedicated handler, and management of *EPC* upon completion. One slight difference is the addition of code to reinitialise *Count* and *Compare* to ensure timer interrupts keep being triggered, but otherwise one would expect Listing 12.2 and Listing 12.6

```
.kdata

id:       .word   id0
          .word   id1
          .word   id2
          .word   id3
          .word   id4
          .word   id5
          .word   id6
          .word   id7
          .word   id8
          .word   id9
          .word   idA
          .word   idB
          .word   idC
          .word   idD
          .word   idE
          .word   idF

save_at:  .word     0

pid:      .word     1
pcb:      .align    2
          .space  164

.ktext 0x80000180

          .set noat
          add     $k0, $0, $at      # save GPR[1]
          sw      $k0, save_at($0)
          .set at

          mfc0    $k0, $13          # get exception cause
          li      $k1, 0x3F
          and     $k0, $k0, $k1
          lw      $k0, id($k0)      # load handler address
          jr      $k0               # jump to handler address

exit_exp: mfc0    $k0, $14
          addi    $k0, $k0, 4
          mtc0    $k0, $14

exit_int: mtc0    $0,  $13          # clear cause

          li      $k0, 4            # set timer limit
          mtc0    $k0, $11
          li      $k0, 0            # set timer count
          mtc0    $k0, $9
          li      $k0, 0xFF01       # enable interrupts
          mtc0    $k0, $12

          .set noat
          lw      $k0, save_at($0) # restore GPR[1]
          add     $at, $0, $k0
          .set at

          eret                      # return from handler
```

Listing 12.6 An example process scheduler (kernel portion).

```
id0:      lw      $k0, pid($0)     # get PCB index
          li      $k1, 2

          beq     $k0, $k1, id0_init

id0_swap: lw      $k0, pid($0)     # get PCB index

          li      $k1, 132         # scale PCB index
          mul     $k0, $k0, $k1
          la      $k1, pcb
          add     $k0, $k0, $k1

          lw      $k1, save_at($0) # save GPR (special case GPR[1])
          sw      $k1,   4($k0)
          sw      $2,    8($k0)    # save GPR (general case)
          sw      $3,   12($k0)
          ...
          sw      $29, 116($k0)
          sw      $30, 120($k0)
          sw      $31, 124($k0)
          mfc0    $k1, $14         # save EPC
          sw      $k1, 128($k0)

          lw      $k0, pid($0) # get PCB index

          li      $k1, 1           # set PCB index
          add     $k0, $k0, $k1
          and     $k0, $k0, $k1
          sw      $k0, pid($0)

          li      $k1, 132         # scale PCB index
          mul     $k0, $k0, $k1
          la      $k1, pcb
          add     $k0, $k0, $k1

          lw      $k1,   4($k0)    # restore GPR (special case GPR[1])
          sw      $k1, save_at($0)
          lw      $2,    8($k0)    # restore GPR (general case)
          lw      $3,   12($k0)
          ...
          lw      $29, 116($k0)
          lw      $30, 120($k0)
          lw      $31, 124($k0)
          lw      $k1, 128($k0)    # restore EPC
          mtc0    $k1, $14

          j       exit_int

id0_init: la      $k0, pcb
          la      $k1, proc_0      # setup pcb_0
          sw      $k1, 128($k0)
          la      $k1, proc_1      # setup pcb_1
          sw      $k1, 260($k0)

          la      $k0, pid         # start proc_0 (setup pcb index)
          sw      $0,    0($k0)
          la      $k1, proc_0      # start proc_0 (setup EPC)
          mtc0    $k1, $14

          j       exit_int

id1:      ...
```

Listing 12.7 An example process scheduler (kernel portion continued).

to be merged into one and roughly represent a single dispatcher capable of dealing with any exception. Listing 12.7 shows the major difference between the two examples, which is the length of the dedicated handler for *Int* (exception number 0) triggered by the timer interrupt. In Listing 12.6 space for two kernel data structures is allocated: `pid` holds the index of the currently executing PCB (0 for `proc_0` and 1 for `proc_1`), `pcb` holds PCB data relating to each process. We take a very limited approach and assume there will only ever be two processes, i.e., those marked `proc_0` and `proc_1`, and allocate space accordingly. Each entry has enough space to save the entire general-purpose register file as well as the *PC* value for each process; there are no priority levels or other meta-data. We initialise `pid` to the value 2 which essentially marks the system as uninitialised.

The code in Listing 12.7 for *Int* (exception number 0) starts by retrieving and testing the value of `pid`. If it finds the system uninitialised, it branches to the label `id0_init` and initialises it. To do this, it stores the addresses of `proc_0` and `proc_1` in the *PC* entries within their respective PCB space and then schedules execution of `proc_0` by setting the `pid` value and updating the *EPC* value to match. If the system is already initialised, one of the processes must already be running and control drops through to the label `id0_swap` to perform a context switch. The handler first loads the current PBC index and scales it to point at the corresponding space within the `pcb` structure. It then saves the context of the current process, storing each general-purpose register and the *PC* value where execution should resume (which is loaded from *EPC*). It then needs to decide which other process should be scheduled for execution: in this example, since there are only two processes the choice is easy ! The handler toggles the current PCB index to the other process before again manipulating the index into a pointer at `pcb`. It then restores the process context, including the *PC* value (which is stored in *EPC*), before finally branching back to the generic handler which returns control to the newly scheduled process.

12.7 Further Reading

- A. Silberschatz, P.B. Galvin and G. Gagne.
  Operating System Concepts.
  John Wiley, 2005. ISBN: 0-471-69466-5.
- A.S. Woodhull and A.S. Tanenbaum.
  Operating Systems Design and Implementation.
  Prentice-Hall, 2006. ISBN: 0-131-42938-8.

12.8 Example Questions

48. Imagine processes that execute on a MIPS32-based processor are scheduled by an operating system kernel. This platform is used in an application that gathers data from a number of sensors. Efficient context switching is important; if a device driver does not begin execution soon after an interrupt, important data could be lost from the associated sensor.

Describe **one** way the processor could be improved to cope better with this demand.

49. In the context of virtual memory as implemented by a given operating system:

a. Describe the resident set versus the working set of a process.
b. What is "thrashing" and how might one detect and resolve the problem ?
c. List some advantages and disadvantages of increasing the page size (given that the physical and virtual address spaces are of fixed size).

50. a. List **two** reasons a process might be terminated.
b. List **two** reasons a process might be suspended.

Chapter 13
Efficient Programming

Rules of Optimization:
Rule 1: Don't do it.
Rule 2 (for experts only): Don't do it yet.

– M. A. Jackson

Premature optimization is the root of all evil.

– C.A.R. Hoare

Abstract In this chapter we give an overview of common program optimisation techniques. In each case the aim is to retain the program semantics but improve the "quality" somehow. For example, one might re-write the program so it executes more quickly or so that it uses less memory. These techniques are notoriously ad-hoc: our aim is to present a reasoned description, making the implications of each case clear when describing what they achieve. Many of the presented techniques can be achieved by hand *or* by using some level of automation in the compiler. In the former case it is important to understand the technique so it can be applied; in the latter case the understanding is important so one can reason about what the compiler is doing and why. The overall aim is that a given program gets the most from the underlying hardware: even efficient processors are only as efficient as the programs they execute !

13.1 Introduction

Given the advice in the quotes above, it is hard to imagine why one would engage in optimisation of a program: such an act seems to fly in the face of modern software engineering practise which preaches scalability, maintainability and portability above all else. However, the advent of structured programming constructs to support this philosophy has produced a new breed of programmer who is somewhat more detached from the underlying hardware than his predecessors. Although languages

like C have clear advantages to assembly language in terms of productivity and portability, they make the execution characteristics less clear. That is, it is not *as* clear what the processor will be doing when executing a C program as it is an assembly language where one clearly has more control. In addition, and even though modern compiler techniques improve all the time, the act of translation is often far from perfect: unconventional or unexpected high-level programs can invoke worst-case behaviour in compilers which fail to generate efficient results. With these issues in mind, use of guidelines [59] and clever programming tricks [64] can improve the performance of high-level programs by, in some sense, presenting a "processor friendly" version of the program in question. Where this can be achieved in a manner that follows software engineering practise, rather than being ad-hoc and undocumented, optimisation can offer a good trade-off between software performance and complexity.

In the early 1900s, Italian economist Vilfredo Pareto observed that 80% of income in Italy was collected by 20% of the population. Although the exact distribution might change this turns out to be a fairly general rule of thumb: a large proportion of the effect is often as a result of a small proportion of the causes. This is certainly true when considering the optimisation of programs; it is very common to find that a small **kernel** is responsible for a large part of execution time for example. Our focus here is on reasoned techniques to optimise C programs and, as such, we specifically target the small kernels of such programs that dominate, namely loops (nests of loops in particular) and function calls.

13.2 "Space" Conscious Programming

13.2.1 Reducing Register Pressure

Within a given processor, the general-purpose register file is finite in size. Use of the registers needs to be managed carefully since if one fills the available space, values will need to be stored in the slower main memory and execution speed will suffer as a result. Modern compilers are quite good at solving the **register allocation** problem: given a set of variables in the program, the compiler assigns them to registers so that a given register is not required to hold more than one live value at a time. However, there might be times when there are no free registers: either the compiler has to spill one of the live values to memory or the programmer must re-write the program somehow to help out.

One well-known, but very specific trick is the swapping of values via XOR. To see that this is possible imagine $a = \alpha$ and $b = \beta$; the following operations are applied:

$$\begin{aligned} a &\leftarrow a \oplus b = \alpha \oplus \beta \\ b &\leftarrow a \oplus b = \alpha \oplus \beta \oplus \beta = \alpha \\ a &\leftarrow a \oplus b = \alpha \oplus \beta \oplus \alpha = \beta \end{aligned}$$

so that afterwards $a = \beta$ and $b = \alpha$, i.e., we have swapped the values. If we represent a and b by the C variables `a` and `b`, we can thus write

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

instead of something like

```
t = a;
a = b;
b = t;
```

which implements the swap using moves and extra variable `t` that would need to be allocated a register.

13.2.2 Reducing Memory Allocation

When languages have dynamic memory allocation, calls to the memory allocator (i.e., the functions that assign blocks of memory to the user) are costly. For example, using `malloc` in C programs can cost orders of magnitude more than an arithmetic operation even though this is somewhat hidden by the language. The opposite operation is equally as costly: de-allocation of memory via a call to `free` can also be expensive. For languages such as Java which support automatic de-allocation via garbage collection the situation is even worse. The support system for such languages will periodically scan for unused memory it has previously allocated and return it to the heap ready for re-allocation. This process presents a further performance penalty.

With this in mind, it can make sense to at least try to minimise the use of the memory allocator by reducing the number of calls to `malloc` and `free`. Consider a running example: a point P in 2D space can be represented by a pair of coordinates x and y to give $P = (x, y)$. A vector in 2D space can be thought of as a line going from the origin $O = (0, 0)$ to the point $V = (x, y)$. Addition of vectors means adding their components so that if $V_1 = (x_1, y_1)$ and $V_2 = (x_2, y_2)$, then $V_3 = V_1 + V_2 = (x_1 + x_2, y_1 + y_2)$. Implementation of this operation in C might be achieved by the following fragment:

```
typedef struct __vector
{
  int x;
  int y;
}
vector;

vector* add( vector* a, vector* b )
{
  vector* r = ( vector* )( malloc( sizeof( vector ) ) );

  r->x = a->x + b->x;
```

```
  r->y = a->y + b->y;

  return r;
}
```

In this case, every time the `add` function is called to add two vectors together, a new `vector` structure will be allocated and returned. This creates two problems from a performance perspective. Firstly, there is an overhead from each call to the memory allocator to create the extra structure. Secondly, there is an overhead since at some later time, the de-allocator (invoked either manually or automatically) will need to spend some time dealing with the extra structure to clean up the heap. Although these overheads might seem small, they can quickly mount up if for example the `add` method is used inside a loop such as

```
vector* sum( vector* v, int n )
{
  vector* r = ( vector* )( malloc( sizeof( vector ) ) );

  for( i = 0; i < n; i++ ) {
    r = add( r, v + i );
  }

  return r;
}
```

The code loops through and accumulates n values in order to perform a vector summation. However, on each loop iteration a new `vector` structure is created by the `add` method which is clearly unsatisfactory from a performance perspective. To improve the performance of the code fragment, we should analyse the use of the `add` method. If we find, for example, that the most often used form of vector addition looks like $a = a + b$, as above, rather than $c = a + b$ (i.e., roughly speaking we are accumulating rather than adding) then we can implement an optimisation whereby the number of allocation calls is significantly decreased:

```
void add( vector* r, vector* a, vector* b )
{
  r->x = a->x + b->x;
  r->y = a->y + b->y;
}

vector* sum( vector* v, int n )
{
  vector* r = ( vector* )( malloc( sizeof( vector ) ) )

  for( i = 0; i < n; i++ ) {
    add( r, r, v + i );
  }

  return r;
}
```

A similar situation exists where we allocate memory within a function but unlike above, do not need to return this to the caller but simply use it internally. Consider a

function which allocates two temporary vectors to start with, then performs some arbitrary calculation before freeing the temporary vectors and returning the calculated result:

```
int foo( ... )
{
  vector* t0 = ( vector* )( malloc( sizeof( vector ) ) );
  vector* t1 = ( vector* )( malloc( sizeof( vector ) ) );

  int r = 0;

  ... calculate r ...

  free( t0 );
  free( t1 );

  return r;
}
```

For a seasoned C programmer this code is clearly poor practise since the two temporary vectors can be created on the stack rather than on the heap; this means the calls to `malloc` can be trivially removed as follows:

```
int foo( ... )
{
  vector t0;
  vector t1;

  int r = 0;

  ... calculate r ...

  return r;
}
```

In this second case, instead of the vectors being created manually at run-time, the compiler incorporates them into the stack frame for the `foo` function and implements their allocation in the function prologue via pushes and pops to and from the stack pointer *SP*. The code therefore performs exactly the same task as before but without the costly calls to the memory allocation system: even freeing the objects is catered for as part of the method exit and resulting function epilogue. The main change is syntactic: access to the vectors now performed using the C dot operator (or direct member access, e.g., `t0.x`) rather than the arrow operator (or member access via pointer, e.g., `t0->x`).

13.3 "Time" Conscious Programming

13.3.1 Effective Short-circuiting

Consider the logical expressions

$$a \wedge b$$

and

$$a \vee b.$$

From Chapter 1 we know that

$$\text{false} \wedge b = \text{false}$$

and

$$\text{true} \vee b = \text{true},$$

i.e., for these cases, the value of b does not matter. Good C compilers use a concept called **short-circuiting** to optimise the performance of programs that contain expressions like this: imagine we write a C program which uses such an expression to control a conditional statement. For example, imagine we write

```
if( a && b ) {
  ...
}
```

such that the statement body is executed if the expression `a && b` evaluates to true (or non-zero), and is not executed otherwise. The idea is that we start by evaluating just `a`. As above, if the result is false, then we know it does not matter what `b` evaluates to: we are going to get false either way. Therefore the compiler arranges to save some time in the event that `a` evaluates to false by skipping over the evaluation of `b`; only if `a` evaluates to true is `b`, and hence the overall expression, evaluated. In this case we used AND, a similar situation exists for OR except that we evaluate `b` only if `a` evaluates to false: if `a` evaluates to true there is no point evaluating `b` since `a || b` will be true either way. Neither case may sound like much of a saving, but remember that where we have used `a` and `b`, one could substitute arbitrarily expressions (as long as they have no side effects) whose cost is potentially more significant.

Although the compiler implements the short-circuiting scheme automatically, as programmers can arrange our code to take advantage of it effectively. For example, in theory there is no difference between the expressions

$$a \wedge b$$

and

$$b \wedge a.$$

In practise, however, there is a difference. Specifically, if short-circuiting is employed and a is likely to be false then the first expression is better than the second

since it represents a situation where we are less likely to have to evaluate b. Moving back to C again, consider a more complicated condition with six terms:

```
if( a && b && c && d && e && f ) {
  ...
}
```

The discussion above essentially tells us that it can pay to think about the probability that one of the terms will evaluate to false, and re-order the condition so those with a high probability are moved to the left and hence evaluated first: this gives a high(er) probability that those to the right will not be evaluated at all and that we improve performance as a result.

13.3.2 Branch Elimination

Pipelined and superscalar processors are most efficient when executing so-called **straight-line code**, i.e., code with no branches in it. As soon as branches are introduced, the processor has to deal with the related control dependency in some way with throughput typically degraded as a result. This is a problem in terms of the use of structured programming where high-level statements such as loops and conditionals, which rely extensively on branches, are used as standard. The (partial) solution is to implement some form of program restructuring, either manually or via the compiler, whereby branches are eliminated. Consider the short conditional statement whereby we set $x = y$ if $c = 1$ and $x = z$ otherwise. In C, using the variables `c`, `x`, `y` and `z` to represent c, x, y and z, we could implement this as

```
if( c ) {
  x = y;
}
else {
  x = z;
}
```

or as the shortened form:

```
x = c ? y : z;
```

Either way, at a machine level the code fragment would clearly need to use branches; assuming `c`, `x`, `y` and `z` are stored in the registers $GPR[2\ldots5]$ a translation into machine-level operations might look something like the following:

$$
\begin{array}{ll}
 & \ldots \\
 & \text{if } GPR[2] = 0, PC \leftarrow skip \\
take: & GPR[2] \leftarrow GPR[3] \\
 & PC \leftarrow done \\
skip: & GPR[2] \leftarrow GPR[4] \\
done: & \ldots
\end{array}
$$

The problem is, assuming we cannot use features such as predicated execution, how can we remove the branch instructions ? At first glance it might seem impossible but there are two potential approaches in this specific example. Both assume that the conditional `c` is 0 when we want to select the false branch of the conditional and 1 otherwise; some compilers will support this, others assume that `c` will just be non-zero rather than 1 in which case our approaches, as stated, fail.

In the first approach, we construct a mask value $m = c - 1$ so that if $c = 0$ then $m = -1$ and if $c = 1$, then $m = 0$. Because of the way the twos-complement representation works, if $c = 0$ then m is a value where all the bits are set to 1 and if $c = 1$ then m is a value where all the bits are set to 0. We can use this mask value to select the right result from y and z using the logical expression

$$x = (m \wedge z) \vee (\neg m \wedge y).$$

That is, if $c = 0$ then m is a value with all bits set to one and we have that $m \wedge z = z$ and $\neg m \wedge y = 0$ so $(m \wedge z) \vee (\neg m \wedge y) = z$. Conversely, if $c = 1$ then m is a value with all bits set to zero and we have that $m \wedge z = 0$ and $\neg m \wedge y = y$ so $(m \wedge z) \vee (\neg m \wedge y) = y$. So we can implement the conditional in C as

```
m = c - 1;
x = ( m & z ) | ( ~m & y );
```

This would take five machine-level operations, one more than the version with branching in, but without branches it will probably run quicker due to the fact it is more friendly to the pipeline.

In the second approach, we use the fact that we can select values at controlled addresses using an indexed addressing mode. Specifically, given a small array `A`, we can implement the conditional in C as the following sequence of assignments

```
A[ 0 ] = z;
A[ 1 ] = y;
x      = A[ c ];
```

If `c` is zero, we set `x` to the contents of `A[0]` which is `z`; if `c` is one we set `x` to the value `A[1]` which is `y`. This time we only use three instructions which sounds great but it is not as clear if this will actually be faster because all three access memory: typically this will be slow, even compared to the impact of branches on the pipeline.

13.3.3 Loop Fusion and Fission

Consider the implementation of a vector addition operation; we want to use two n-element vectors B and C to compute

$$A_i = B_i + C_i,\ 0 \leq i < n.$$

This operation might be implemented as follows:

```
for( int i = 0; i < n; i++ ) {
  A[ i ] = B[ i ] + C[ i ];
}
```

where the C variables `A`, `B`, `C`, `i` and `n` represent A, B, C, i and n. An **induction variable** is a term referring to a variable that is incremented or decremented in every iteration of the loop; in this example `i` is the induction variable. The use of a loop makes implementing the required behaviour simple and concise. However, there is an overhead associated with this expressiveness: for every execution of the loop body we need to execute the associated loop control instructions. This overhead can be costly. In this case, the loop body is just four instructions (two loads, an addition and a store); ignoring the initialisation of i, the loop control introduces (at least) two extra instructions (increment and test of loop counter) per-iteration. So ignoring the impact of the branch on the pipeline, roughly a third of all computation is simply overhead.

The technique of **loop fusion** aims to reduce this overhead in particular situations. The basic idea is when two similar loops exist, one could try to share the overhead between them by joining or fusing the loops together. Following on from above, consider an example where we have two n-element vectors B and C and for some application need their sum and dot-product; i.e., we need to compute

$$A_i = B_i + C_i,\ 0 \leq i < n$$

and

$$a = \sum_{i=0}^{n-1} B_i \cdot C_i.$$

The first operation is already implemented above, the second can be implemented as follows:

```
a = 0;

for( int i = 0; i < n; i++ ) {
  a += B[ i ] * C[ i ];
}
```

where the C variables `A`, `B`, `C`, `a`, `i` and `n` represent A, B, C, a, i and n. Notice that both loops iterate over the same values of i, the induction variable, that runs through $0 \leq i < n$. It therefore makes perfect sense to fuse the two loops into one thus saving the cost of the overhead; the fused result can be realised as follows:

```
int a = 0;

for( int i = 0; i < n; i++ ) {
  int b = B[ i ];
  int c = C[ i ];

  A[ i ]  = b + c;
  a      += b * c;
}
```

Such code might appear almost moronic in that the proposed optimisation is so obvious. However, other issues might cloud the fact that we can perform the fusion and make advantages less obvious. For example, if the vector addition and dot-product functionality were implemented as functions, we might not actually *see* the two loops alongside each other in code; the topic of function inlining described in the next section might uncover the potential for optimisation in this case. The second advantage of performing the fusion of the two loops is that we can now perform some common sub-expression elimination that might not have been possible before. In this particular case, both loops load the values `B[i]` and `C[i]`: clearly we only need to do this once and can share the result thus saving around $2n$ load operations as well as the loop overhead.

It is not always clear-cut that loop fusion is a good idea. Sometimes the opposite approach of **loop fission**, e.g., splitting one loop into two, is the more effective choice. Consider the following contrived program fragment as an example:

```
for( int i = 0; i < n; i++ ) {
  A[ i ] = A[ i ] + 1;
  B[ i ] = B[ i ] + 1;
}
```

It performs the component-wise increment of two n-element vectors called A and B that are represented by the C variables `A` and `B`. Using common sense, the fragment adopts the fusion approach and merges the two operations into one loop. But consider what happens if `A` and `B` are mapped to the same place in the cache. Access to `A[0]` provokes a cache-miss but then `B[0]` is loaded, again missing and evicting `A[0]`. Then `A[1]` is accessed and would have provoked a cache-hit if it were not for the fact that `B[0]` had evicted it. So every iteration of the loop we are losing out because the locality of accesses in `A` and `B` is not being capitalised on. To remedy this situation, we can split the loop into two:

```
for( int i = 0; i < n; i++ ) {
  A[ i ] = A[ i ] + 1;
}

for( int i = 0; i < n; i++ ) {
  B[ i ] = B[ i ] + 1;
}
```

Note that in this particular case, loop fission has taken one loop and split it into two by splitting the loop *body*. An alternative approach, which is useful later, is to keep the same loop body but split the loop *range* instead. Instead of having the loop counter range over $0 \leq i < n$, we might for example select some $0 < m < n$ and produce two loops: one loop has a loop counter range over $0 \leq i < m$ and the other has a loop counter range over $m \leq j < n$. Using the above as an example, this would yield

```
for( int i = 0; i < m; i++ ) {
  A[ i ] = A[ i ] + 1;
  B[ i ] = B[ i ] + 1;
}
```

```
for( int j = m; j < n; j++ ) {
  A[ j ] = A[ j ] + 1;
  B[ j ] = B[ j ] + 1;
}
```

13.3.4 Loop Unrolling

The technique of **loop unrolling** addresses a similar problem as loop fusion in the sense that both aim to eliminate the overhead associated with operating a given loop. In cases where the number of times the loop will iterate (which is called the **loop bound**) is statically known, one can enumerate all possible values of the induction variable. Then one can simply replicate the loop body that many times and substitute in the enumerated values before removing the loop. Starting with the vector addition implementation from above, i.e.,

```
for( int i = 0; i < n; i++ ) {
  A[ i ] = B[ i ] + C[ i ];
}
```

imagine that in a specific application we always know $n = 8$, and hence that $0 \leq i < 8$. We can unroll the loop to produce

```
A[ 0 ] = B[ 0 ] + C[ 0 ];
A[ 1 ] = B[ 1 ] + C[ 1 ];
A[ 2 ] = B[ 2 ] + C[ 2 ];
A[ 3 ] = B[ 3 ] + C[ 3 ];
A[ 4 ] = B[ 4 ] + C[ 4 ];
A[ 5 ] = B[ 5 ] + C[ 5 ];
A[ 6 ] = B[ 6 ] + C[ 6 ];
A[ 7 ] = B[ 7 ] + C[ 7 ];
```

which does the same thing as the original loop (for this particular n) but has no associated overhead. Usually, loop unrolling is applied to improve performance: in this case we save the cost associated with the loop overhead by removing it entirely. However, there are also some drawbacks in that the generated code size can increase significantly depending on the value of n. Furthermore, one could argue the overall maintainability is decreased; for example if we unroll the loop for $n = 4$, and want to change to $n = 8$ we need to do more work than simply altering the loop bound. Although loop unrolling can be performed by hand, it is generally better to let the compiler do the work since it can evaluate the side effects and relative performance implications.

As well as removing the loop overhead, loop unrolling can be useful to expose operations that can be performed in parallel. This can be achieved using **partial unrolling** whereby instead of unrolling the loop entirely, we just unroll it a bit and reduce the number of iterations; consider partially unrolling the example above by a factor of four:

```
for( int i = 0; i <= 4; i += 4 ) {
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
  A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}
```

By performing this partial unrolling, it is now clear that the four operations in the loop body can be performed in parallel; before the unrolling this fact was obscured by the loop construct. Of course, we have assumed that the loop bound is known and divisible by the factor by which we unroll the loop (e.g., four does divide eight). If we cannot make this assumption, we need to add a **loop prologue** (resp. **loop epilogue**) that deals with extra iterations before (resp. after) the loop body. For example, imagine we wanted to partially unroll the loop above by a factor of three instead of four; clearly three does not divide eight so we are forced to construct an epilogue which deals with the final two elements:

```
for( int i = 0; i <= 5; i += 3 ) {
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
}

A[ 6 ] = B[ 6 ] + C[ 6 ];
A[ 7 ] = B[ 7 ] + C[ 7 ];
```

As an aside, note that one *could* view the above as two transformations: a loop fission to split the original loop into two parts (the partially unrolled part and the loop epilogue) followed by a full unrolling of the epilogue. Clearly we could have retained the loop epilogue as a "real" loop if this was beneficial: for example if the loop bound for the epilogue (i.e., the number of elements dealt with by the epilogue) had been large, unrolling it would have been unattractive.

13.3.5 Loop Hoisting

The body of a loop is potentially executed many times. As such, we want to avoid including **loop invariant** code in the loop body. Imagine there is some arithmetic operation n which takes the general form $x \leftarrow y \odot z$. That is, it takes y and z as operands then applies some operation $\odot$ (e.g., an addition) to produce x as the result. The operation n is termed loop invariant if

1. each of y and z is constant, or
2. all definitions of y or z that reach n are outside the loop, or
3. only one definition of y or z reaches n, and that definition is loop invariant.

If these conditions hold, we can think about **hoisting** the operation out of the loop. Consider the following contrived program fragment as an example:

```
b = 10;
c = 20;

for( int i = 0; i < n; i++ ) {
  A[ i ] = B[ i ] + b + C[ i ] + c;
}
```

In this case, the addition of `b` and `c` clearly gets executed in every loop iteration. However, since both `b` and `c` are constant, the addition is loop invariant and can hence be hoisted out of the loop without any change to the meaning:

```
b = 10;
c = 20;

t = b + c;

for( int i = 0; i < n; i++ ) {
  A[ i ] = B[ i ] + C[ i ] + t;
}
```

As an aside, a similar case can apply where a conditional statement exists inside the loop body. Clearly, the condition will be tested in every loop iteration; if it is only satisfied rarely, this is a waste. Sometimes we can hoist the conditional outside the loop to avoid overhead. Again consider a contrived example program fragment:

```
for( int i = 0; i < n; i++ )
{
  if( i == 0 ) {
    A[ i ] = B[ i ] + C[ i ];
  }
  else {
    A[ i ] = B[ i ] - C[ i ];
  }
}
```

In this case we can apply loop fission to break the loop into two parts:

```
A[ 0 ] = B[ 0 ] + C[ 0 ];

for( int j = 1; j < n; j++ ) {
  A[ j ] = B[ j ] - C[ j ];
}
```

Actually, the first part is just a single operation rather than a loop so one might argue we have simultaneously applied loop fission (to break the loop into two ranges of $i = 0$ and $1 \leq j < n$) and then loop unrolling (to turn the loop over the range $i = 0$ into a single assignment). Either way one can see that the conditional statement has been eliminated, so performance is improved.

13.3.6 Loop Interchange

Consider two $(n \times m)$-element matrices B and C and an operation to compute a third matrix A as the component-wise addition of B and C:

$$A_{i,j} = B_{i,j} + C_{i,j},\ 0 \leq i < n,\ 0 \leq j < m.$$

One can imagine two different ways to implement this operation. For example the two loop nests

```
for( int j = 0; j < m; j++ ) {
  for( int i = 0; i < n; i++ ) {
    A[ i ][ j ] = B[ i ][ j ] + C[ i ][ j ];
  }
}
```

and

```
for( int i = 0; i < n; i++ ) {
  for( int j = 0; j < m; j++ ) {
    A[ i ][ j ] = B[ i ][ j ] + C[ i ][ j ];
  }
}
```

both do the same thing even though the first iterates in **row-major** order whereas the second uses **column-major** order (in the sense that the inner loop iterates over rows or columns of the matrices respectively). However, depending on the memory layout for `A`, `B` and `C` one might be more attractive than the other. For example, C defines arrays in memory to be row-major in the sense that consecutive elements in memory are those in the same row (i.e., in consecutive columns); Fortran defines **column-major** arrays where the opposite is true. Put more simply, if one defined the row-major C array:

```
uint8 A[ 3 ][ 2 ] = { { 0, 1 },
                      { 2, 3 },
                      { 4, 5 } };
```

then one would expect the byte-addressed memory content to be

$$\begin{aligned} MEM[\&\mathtt{A}+0] &= 0 \\ MEM[\&\mathtt{A}+1] &= 1 \\ MEM[\&\mathtt{A}+2] &= 2 \\ MEM[\&\mathtt{A}+3] &= 3 \\ MEM[\&\mathtt{A}+4] &= 4 \\ MEM[\&\mathtt{A}+5] &= 5 \end{aligned}$$

whereas if a column-major order was used, then the memory content would be

$$\begin{aligned} MEM[\&\mathtt{A}+0] &= 0 \\ MEM[\&\mathtt{A}+1] &= 2 \\ MEM[\&\mathtt{A}+2] &= 4 \\ MEM[\&\mathtt{A}+3] &= 1 \\ MEM[\&\mathtt{A}+4] &= 3 \\ MEM[\&\mathtt{A}+5] &= 5. \end{aligned}$$

The problem is, if a row-major memory layout is used but then we iterate through it in column-major order, then the locality of our accesses is worse than if we iterated through it in row-major order. In the example above, using a row-major memory layout and column-major iteration order we access $MEM[\&\mathtt{A}+0]$ then $MEM[\&\mathtt{A}+3]$ and so on whereas to get the best use of our cache, we would prefer to have accessed $MEM[\&\mathtt{A}+0]$ then $MEM[\&\mathtt{A}+1]$ and so on since these accesses are more spatially local.

Since the latency associated with accessing main memory is large, it is often attractive to restructure a program to improve cache performance. Automation of such restructuring has been investigated for both data [10, 16] and instructions [26, 51]. In this context, if we start with a loop nest which uses row-major (resp. column-major) iteration order, we can sometimes transform it into a column-major (resp. row-major) version which has the same meaning but exhibits better cache locality. This is termed **loop interchange**; essentially we are interchanging two loops (as above) as long as doing so does not cause problems in terms of dependency between operations within the loop bodies.

13.3.7 Loop Blocking

Another example of cache conscious program restructuring is the technique of **loop blocking**. This technique can be applied to loop nests that iterate over elements of, for example, a large array: the basic idea is to iterate over small blocks of the array at a time, rather than the whole array in one go. As an example, consider the multiplication of two $(n \times n)$-element, i.e., square, matrices B and C to produce A where

$$A_{i,j} = \sum_{k=0}^{n-1} B_{i,k} \cdot C_{k,j},\ 0 \leq i, j < n.$$

This operation can be implemented as follows:

```
for( int i = 0; i < n; i++ ) {
  for( int j = 0; j < n; j++ ) {
    A[ i ][ j ] = 0;
  }
}

for( int i = 0; i < n; i++ ) {
  for( int j = 0; j < n; j++ ) {
    for( int k = 0; k < n; k++ ) {
```

```
      A[ i ][ j ] += B[ i ][ k ] * C[ k ][ j ];
    }
  }
}
```

with the C variables `A`, `B`, `C`, `i`, `j`, `k` and `n` representing A, B, C, i, j, k and n. The loop nest iterates over the $n \times n$ elements of A, B and C in the inner loop. The problem is, as n grows large, perhaps none of the matrices will fit entirely into the cache. This results in the cache being ineffective in utilising the available locality. For example, the loop nest iterates over the entirety of B, using the reference $C_{k,j}$, for each iteration of the outer loop. As such, we would like C to remain in the cache since we are reusing the content on each iteration of said loop.

The general strategy is to consider each loop in the loop nest above and to apply a generic **loop blocking** transformation from one loop

```
for( int i = 0; i < n; i++ ) {
  ...
}
```

into another

```
for( int ii = 0; ii < n; ii += b ) {
  for( int i = ii; i < min( n, ii + b ); i++ ) {
    ...
  }
}
```

Both cases iterate over the same (empty) loop body; the loop counter `i` ranges over the same values in each case. However, the first case loops over the whole range of `i` in one go whereas the second breaks the range into smaller blocks of size `b`. Applying the transformation to each loop, we end up with the fairly unpleasant looking alternative:

```
for( int i = 0; i < n; i++ ) {
  for( int j = 0; j < n; j++ ) {
    A[ i ][ j ] = 0;
  }
}

for( int ii = 0; ii < n; ii += b ) {
  for( int i = ii; i < min( n, ii + b ); i++ ) {
    for( int jj = 0; jj < n; jj += b ) {
      for( int j = jj; j < min( n, jj + b ); j++ ) {
        for( int kk = 0; kk < n; kk += b ) {
          for( int k = kk; k < min( n, kk + b ); k++ ) {
            A[ i ][ j ] += B[ i ][ k ] * C[ k ][ j ];
          }
        }
      }
    }
  }
}
```

Next we apply the idea of loop interchange to rearrange the control loops to the outside of the nest and the block loop to the inside:

```
for( int i = 0; i < n; i++ ) {
  for( int j = 0; j < n; j++ ) {
    A[ i ][ j ] = 0;
  }
}

for( int ii = 0; ii < n; ii += b ) {
  for( int jj = 0; jj < n; jj += b ) {
    for( int kk = 0; kk < n; kk += b ) {
      for( int i = ii; i < min( n, ii + b ); i++ ) {
        for( int j = jj; j < min( n, jj + b ); j++ ) {
          for( int k = kk; k < min( n, kk + b ); k++ ) {
            A[ i ][ j ] += B[ i ][ k ] * C[ k ][ j ];
          }
        }
      }
    }
  }
}
```

In doing so, we split the iteration into smaller blocks: instead of working over the $n \times n$ elements in the inner loop, we work over $b \times b$ elements. Although there is massively more overhead in operating this code in terms of the loop control, as long as these smaller $b \times b$ blocks of matrix fit into the cache, performance can still be vastly improved. For example, if the block of C now fits into the cache, accesses to $C_{k,j}$ will always provoke cache-hits which are vastly less expensive than cache-misses.

13.3.8 Function Inlining

We have already discussed how the compiler uses a stack to implement function calls. Ignoring recursive functions, the advantages of decomposing a program in to small functions are mainly related to modularity: it becomes easier to locate and correct programming errors. The disadvantage of this approach is that there is a cost or overhead associated with functions in that it takes instructions, and hence time, to implement the function prologue and epilogue as well as the actual branch. In a sense, a function call is entirely overhead since it is not doing any useful computation in itself.

To address this issue, one typically uses **function inlining** to restructure the program into a more efficient version. Consider the example of a simple function to sum the elements in an n-element vector A, i.e., compute

$$s = \sum_{i=0}^{n-1} A_i.$$

Taking an excessively strict approach to modularity, an implementation of the function might look like the following:

```
inline int add( int x, int y )
{
  return x + y;
}

int sum( int* A, int n )
{
  int s = 0;

  for( int i = 0; i < n; i++ ) {
    s = add( s, A[ i ] );
  }

  return s;
}
```

where the C variables `A`, `s`, `i` and `n` represent A, s, i and n. The `sum` function performs calls to the `add` function which, very simply, just adds two inputs x and y together and returns the result. Clearly, the overhead associated with calling `add` is large in comparison to execution of the actual function body. To remove this overhead, we inline `add` into `sum` to give the more sensible implementation:

```
int sum( int* A, int n )
{
  int s = 0;

  for( int i = 0; i < n; i++ ) {
    s = s + A[ i ];
  }

  return s;
}
```

This transformation is performed using roughly similar steps as those in loop unrolling: first we take a copy of the function body and give all the local variables unique names; we then map the function call arguments to the definition parameters and then put the new function body in the place of the function call. Notice that the definition of `add` includes the C `inline` keyword. This signals, or hints, to the compiler that the function should be inlined. Of course, the process can also be done manually.

The clear advantage after the program is restructured is that the cost of the function prologue and epilogue are eliminated. For small values of n this will save time but will perhaps not be too noticeable; for large values of n the saving will be significant. In addition ,inlining can create the opportunity for further optimisations, for example common sub-expression elimination, which would have been masked beforehand. However, there are disadvantages to inlining, the most serious of which is the potential for **code bloat**. The whole idea of a function is that there need only be one block of code that is called from multiple locations. By inlining the function, all these locations now replicate the code. This typically leads to the executable program being much larger in size which can be a problem. Finally, like loop unrolling, it can lead to problems of maintainability if performed manually: changes to a func-

tion manually inlined in several locations requires more work than changing just one function.

There are some more intricacies as well: consider the example of computing the factorial function which is defined by

$$\textsc{Fac}(x) = x \cdot (x-1) \cdot (x-2) \cdots 2 \cdot 1.$$

so that $\textsc{Fac}(5) = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ for example. The standard method of implementing this is via a recursive function:

```
int fac( int x )
{
  if( x == 1 ) {
    return 1;
  }
  else {
    return x * fac( x - 1 );
  }
}
```

In the context of function inlining this is a problem. That is, we cannot inline the `fac` function because it contains a call to itself; how would we know when to stop inlining given an unknown input ? However, the `fac` function has a special property in that in the recursive case, the call to `fac` is the last thing to be computed. Functions of this form are termed **tail-end recursive** and can be converted into an iterative form. This transformation, which can be applied either manually or automatically by most compilers, still removes function call overhead in terms of execution time. Perhaps more importantly however, is also vastly reduces amount of stack space used: in the recursive version a new stack frame would be required for each recursive call made. After the tail-end recursion is inlined, the implementation becomes somewhat similar to the following:

```
int fac( int x )
{
  int t = 1;

  while( 1 ) {
    if( x == 1 ) {
      return t;
    }

    t = x * t;
    x = x - 1;
  }
}
```

It uses the variable `t` to represent the return value from the recursive call; the initial assignment represents the base case of the recursion, the assignment inside the loop represents the recursive case. Notice that we also keep track of `x` which represents the argument to the recursive call; using the value we know when to exit the iterative loop representing the act of recursion.

13.3.9 Software Pipelining

In a hardware pipeline (e.g., a pipelined processor), execution of different instructions is overlapped. For example, the fetch of one instruction happens at the same time as another is being decoded: their execution is overlapped. The technique of **software pipelining** is another, perhaps more advanced means of optimising loops with a specific form. In a software pipeline, the execution different code sequences (most commonly loop iterations) is overlapped with a view to breaking dependencies which could otherwise cause hazards, and hence stalls, within the processor. Consider the contrived example loop:

```
for( int i = 0; i < 9; i++ ) {
  A[ i ] = A[ i ] + 3;
}
```

Implementation of the loop body (minus loop control and addressing) will consist of three instructions: a load, an addition and a store. Note that these instructions are dependent in the sense that they must be executed sequentially; for example the addition cannot execute before the load has completed.

Abusing notation slightly to permit t_i to denote a temporary symbol in the i-th iteration of the loop, and reading from left to right and top to bottom, from the processors perspective the instructions executed form the sequence:

$$\begin{array}{lll} t_0 \leftarrow MEM[\&\mathtt{A}+0] & t_0 \leftarrow t_0+3 & MEM[\&\mathtt{A}+0] \leftarrow t_0 \\ t_1 \leftarrow MEM[\&\mathtt{A}+1] & t_1 \leftarrow t_1+3 & MEM[\&\mathtt{A}+1] \leftarrow t_1 \\ t_2 \leftarrow MEM[\&\mathtt{A}+2] & t_2 \leftarrow t_2+3 & MEM[\&\mathtt{A}+2] \leftarrow t_2 \\ \vdots & \vdots & \vdots \end{array}$$

This is a problem for superscalar processors where we aim to issue one instruction (or more) per-cycle. Imagine each of the instructions has a latency of three cycles, i.e., it takes three cycles from an instruction being issued to when it make a result available. Each iteration of the loop takes $3+3+1=7$ cycles to complete (assuming the load in the next iteration can overlap the store in the current one). The problem is that the *distance* between dependent operations is less than the *latency* to execute those operations; this means they stall while waiting for the dependency to be resolved. For example, there is just one cycle between when t_0 is loaded from $MEM[\&\mathtt{A}+0]$ to when t_0 is incremented so the addition instruction must stall until the load produces a result.

However, a given iteration of the loop is independent from every other; for example iteration zero can execute at the same time as iteration one and so on. Imagine we partially unroll the loop (we might still need a loop prologue and epilogue but ignore this for now):

```
for( int i = 0; i <= 6; i += 3 ) {
  A[ i+0 ] = A[ i+0 ] + 3;
  A[ i+1 ] = A[ i+1 ] + 3;
  A[ i+2 ] = A[ i+2 ] + 3;
}
```

At face value, nothing has changed. But looking at the actual instructions the processor would execute, we can now re-schedule them so the processor instead executes the sequence:

$$\begin{array}{ccc} t_0 \leftarrow MEM[\&\mathtt{A}+0] & t_1 \leftarrow MEM[\&\mathtt{A}+1] & t_2 \leftarrow MEM[\&\mathtt{A}+2] \\ t_0 \leftarrow t_0+3 & t_1 \leftarrow t_1+3 & t_2 \leftarrow t_2+3 \\ MEM[\&\mathtt{A}+0] \leftarrow t_0 & MEM[\&\mathtt{A}+1] \leftarrow t_1 & MEM[\&\mathtt{A}+2] \leftarrow t_2 \\ \vdots & \vdots & \vdots \end{array}$$

Unlike the initial sequence, the processor is now free to issue one operation per-cycle since the distance between each dependent operation matches their latency. For example, there are now three cycles between when t_0 is loaded from $MEM[\&\mathtt{A}+0]$ to when t_0 is incremented. As a result, each loop iteration takes only $1+1+1=3$ cycles !

Things can get more tricky than this: imagine we replace the addition with a multiplication that takes 6 cycles to execute rather than 3:

```
for( int i = 0; i < 9; i++ ) {
  A[ i ] = A[ i ] * 3;
}
```

Our original software pipeline does not function correctly; the distance between the multiplication and the store is less than than multiplication latency. But imagine this (at face value, quite bizarre) rescheduling of instructions in the middle portion of the loop:

$$\begin{array}{ccc} \vdots & \vdots & \vdots \\ t_3 \leftarrow MEM[\&\mathtt{A}+3] & t_2 \leftarrow t_2 \cdot 3 & MEM[\&\mathtt{A}+0] \leftarrow t_0 \\ t_4 \leftarrow MEM[\&\mathtt{A}+4] & t_3 \leftarrow t_3 \cdot 3 & MEM[\&\mathtt{A}+1] \leftarrow t_1 \\ t_5 \leftarrow MEM[\&\mathtt{A}+5] & t_4 \leftarrow t_4 \cdot 3 & MEM[\&\mathtt{A}+2] \leftarrow t_2 \\ t_6 \leftarrow MEM[\&\mathtt{A}+6] & t_5 \leftarrow t_5 \cdot 3 & MEM[\&\mathtt{A}+3] \leftarrow t_3 \\ t_7 \leftarrow MEM[\&\mathtt{A}+7] & t_6 \leftarrow t_6 \cdot 3 & MEM[\&\mathtt{A}+4] \leftarrow t_4 \\ t_8 \leftarrow MEM[\&\mathtt{A}+8] & t_7 \leftarrow t_7 \cdot 3 & MEM[\&\mathtt{A}+5] \leftarrow t_5 \\ \vdots & \vdots & \vdots \end{array}$$

Notice that, for example, the distance between multiplying t_3 and storing the result (more) closely matches the instruction latencies. Of course the corresponding C program makes much less sense, however performance can again be vastly improved due to elimination of dependency related stalls. The only final problem to overcome is that now we need a loop prologue (resp. epilogue). That is, we need to fill (resp. empty) the pipeline in some sense:

$$
\begin{array}{lll}
t_0 \leftarrow MEM[\&\mathtt{A}+0] & & \\
t_1 \leftarrow MEM[\&\mathtt{A}+1] & t_0 \leftarrow t_0 \cdot 3 & \\
t_2 \leftarrow MEM[\&\mathtt{A}+2] & t_1 \leftarrow t_1 \cdot 3 & \\
t_3 \leftarrow MEM[\&\mathtt{A}+3] & t_2 \leftarrow t_2 \cdot 3 & MEM[\&\mathtt{A}+0] \leftarrow t_0 \\
\quad\vdots & \quad\vdots & \quad\vdots \\
t_8 \leftarrow MEM[\&\mathtt{A}+8] & t_7 \leftarrow t_7 \cdot 3 & MEM[\&\mathtt{A}+5] \leftarrow t_5 \\
 & t_8 \leftarrow t_8 \cdot 3 & MEM[\&\mathtt{A}+6] \leftarrow t_6 \\
 & & MEM[\&\mathtt{A}+7] \leftarrow t_7 \\
 & & MEM[\&\mathtt{A}+8] \leftarrow t_8
\end{array}
$$

Depending on the latencies and so on, this can make the prologue (resp. epilogue) quite long.

13.4 Example Questions

51. The following questions ask you to write C functions that operate on unsigned 16-bit integer values represented by the `uint16` type. Each function should be fairly short, i.e., roughly 10–15 lines or fewer; a more efficient function will gain more marks. For each answer, briefly state any advantages and disadvantages of your approach in comparison to alternatives.

a. Without using the operations $<$, $=$, or $>$ or any branch or conditional operations, write a C function that replicates the behaviour of

```
uint16 choose( uint16 c, uint16 x,  uint16 y )
{
  if( c ) {
    return x;
  }
  else {
    return y;
  }
}
```

where you can assume c is used as a Boolean variable, i.e., it always equals either 0 or 1.

b. Without using the multiply operation, write a C function to implement the multiplication $x \cdot 15$ for an unsigned 16-bit integer x.

c. The population count operation takes an input x and computes $H(x)$, the number of bits in the representation of x which are set to 1. Write a C function to compute $H(x)$ for an unsigned 16-bit integer x.

d. Pick one of your solutions from the previous question and show how to accelerate it using the concept of pre-computation; you should write one C function to perform any pre-computation, and one C function that uses the pre-computed data to implement the original operation.

52. Assume that unsigned 16-bit and 8-bit integer values are represented by the `uint16` and `uint8` types. The C function below performs "packed" division by two on elements of a vector x: it takes each 16-bit element $x[i]$, unpacks it into two 8-bit values a and b, divides a and b by two, and packs them back into the 16-bit result:

```
void packed_div2( uint16* x )
{
  for( int i = 0; i < 3; i++ ) {
    uint8 a = ( x[ i ]      ) & 0xFF;
    uint8 b = ( x[ i ] >> 8 ) & 0xFF;

    a = a / 2;
    b = b / 2;

    x[ i ] = ( uint16 )( a      ) |
             ( uint16 )( b << 8 ) ;
  }
}
```

Several optimisation techniques could make this function execute faster:

a. Off-line pre-computation.
b. Specialisation.
c. Parallelism.
d. Program restructuring.

Show how the function could be re-written (using pseudo-code where appropriate) utilising each of these techniques.

53. Consider the following C fragment which adds together two n-element vectors B and C to produce a result A where each element is a 32-bit integer:

```
for( int i = 0; i < n; i++ ) {
  A[ i ] = B[ i ] + C[ i ];
}
```

a. Describe **two** reasons why the use of loop unrolling might increase performance of the fragment, and **one** reason it might decrease performance.
b. Sometimes loop unrolling requires a loop prelude and/or epilogue. Using an example, describe what these are and why they might be required.

Part IV
Appendices

Part IV
Appendices

Appendix A
SPIM: A MIPS32 Simulator

A.1 Introduction

Often it is convenient to write programs for a processor one does not have physical access to. In this sort of situation, use of a **simulator** or **emulator** can plug the resulting gap. Roughly speaking both are software programs that attempt to replicate the behaviour of the related processor. They allow the execution of programs on a virtual processor in some sense. More strictly, an emulator is often defined as simply emulating behaviour in the sense that to software running on it, the emulator *appears* to be the real processor. Underneath the surface however, the emulator can take simplifying steps in providing the illusion of features existing. In contrast, a simulator worries about *exactly* reproducing these features in detail. For example, a simulator might exactly model the ALU of a processor while an emulator only needs to give the illusion that the real ALU is being modelled; it is free to use the ALU provided by the host system rather than emulating that of the system being emulated.

Issues of exact definition aside, **SPIM** is a superb MIPS32 simulator written by James Larus [35]. In the past SPIM focused on modelling the MIPS R2000 processor but now boasts impressively complete support for the full MIPS32 ISA. This, along with the fact that the software is available under a fairly permissive license, makes it ideal for learning MIPS32 programming. You can download SPIM from

`http://pages.cs.wisc.edu/~larus/spim.html`

In this Appendix we cover the features and operation of SPIM version 7.3 that relate to our discussion of the MIPS32 ISA in Chapter 5 and others. We omit details, such as floating point operations, which are not relevant in this context.

A.2 Configuring SPIM

Before one starts to use SPIM as a means of simulating MIPS32 programs, it is important to understand how SPIM will perform this task and how it can be configured to alter various behavioural characteristics. These characteristics are important: without understanding what SPIM should do when used in a certain way, it can be frustrating if the result is not what was expected. Firstly, there are some important facts about what SPIM can and cannot, and will and will not do:

- SPIM does not execute machine code programs. Instead, the front-end can be thought of more like an assembler. One feeds SPIM a MIPS32 assembly language program (with a predefined entry point) which is then executed.
- SPIM does not load the input program at address zero like we assumed in the basic assembler presented in Chapter 10. Instead it loads the program in a way which lies somewhat closer to how an operating system would deal with it, i.e., by placing it at some address in memory and relocating global symbols relative to this.
- As a result of the positioning of SPIM as somewhere between a processor and operating system, it does not simulate low-level architectural features such as the effects of cache memory. However, it does support a limited range of system calls which are available via the `syscall` instruction and which allow programs running on SPIM to interface with the host system. For example they can read input from the keyboard or print results to the console.
- The endianness of the processor SPIM simulates matches that of the host system. That means that the endianness might differ depending on where the SPIM simulator is executed. Therefore, although SPIM itself is portable, programs that run on it are not platform independent.
- The SPIM front-end strictly enforces the reservation of, for example, $GPR[1]$ or `$1` for use in the assembly process; it will not accept input programs which target `$1` explicitly. This also applies to other reserved registers. By reserving this register for it's own use, SPIM ensures, for example, that it can synthesise any pseudo-instructions. So do not be surprised if, during the execution of a program, executed instructions do not exactly match what you would expect: probably SPIM has translated and expanded what you wrote into something more suitable.

Next, note that the behaviour of SPIM can be altered by including one or more command line options when invoking it:

- The `-pseudo` and `-nopseudo` options allow the user to selectively turn on or off the facility for pseudo-instructions during the parsing of programs by the SPIM front-end. The default assumes the `-pseudo` option has been selected to allow pseudo-instructions.
- We discussed the issue of branch delay slots in Chapter 9 and used this concept explicitly when describing a mechanism for function calls in Chapter 11. However, a real MIPS32 processor has other delay slots of this type (for example load and store delay slots). The `-delayed_branches` and `-delayed_loads`

options allow the user to selectively expose programs to the raw processor in terms of these types of feature. The default assumes that neither `-delayed_branches` or `-delayed_loads` has been selected so that neither branch or load delay slots are simulated by SPIM but rather hidden from the user.

- The `-quiet` and `-noquiet` options turn warning messages, which occur when an exception is raised, on and off. The default assumes that the `-noquiet` option has been selected to turn such warnings on.
- The `-bare` and `-asm` options allow the user to switch SPIM between basic and extended modes. The basic mode implies the `-pseudo` and `-noquiet` options and turns off the simulation of branch and load delay slots; this is assumed by default. The extended or bare mode implies all of `-nopseudo`, `-quiet`, `-delayed_branches` and `-delayed_loads` to present a more raw processing environment to the user.
- The `-exception` and `-noexception` options instructs SPIM to either load the default exception handler, or to skip this step. The `-exception_file` option can be used to load a exception handler other than the default from a file. The default exception handler includes basic functionality which simply prints information when exceptions occur, and code at the default SPIM entry point, labelled `__start`, which calls code labelled `main`. Without the default exception handler, the user must provide their own `__start` entry point or explicitly specify an initial address when executing their program.
- The `-mapped_io` and `-nomapped_io` instruct SPIM to turn the memory mapped input/output functionality on or off. Roughly speaking, this turns on or off software-based interrupts generated by SPIM which include timer events and the interface with a terminal type device provided by receiver and a transmitter interrupts. The default assumes the `-nomapped_io` option has been selected to specify turn off memory mapped input/output.

A.3 Controlling SPIM

The default interface to SPIM is via a command line similar in style to the debugger interfaces normally offered for physical processors. After execution from the shell prompt, the user is presented with the SPIM prompt `(spim)` which invites entry of commands to operate the simulator. Listing A.1 shows the invocation of SPIM and the output of issuing the `help` command. Notice that when invoking SPIM, we were careful to specify the command line options `-asm`, `-exception` and `-nomapped_io` so we are clear how the simulator should behave; as the first step, SPIM loads the default exception handler as a result of our choices. The `help` command prints a screen of information about possible commands; we present an explanation of these commands in a logical order (rather than in the same order as the help screen):

- The `exit` and `quit` commands both terminate the current simulator instance, returning the user to the command prompt from where SPIM was executed.

```
bash$ spim -asm -exception -nomapped_io
SPIM Version 7.3. of August 28, 2006
Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/spim/lib/exceptions.s
(spim) help

SPIM is a MIPS R2000 simulator.
Its top-level commands are:
exit  -- Exit the simulator
quit  -- Exit the simulator
read ``FILE'' -- Read FILE of assembly code into memory
load ``FILE'' -- Same as read
run <ADDR> -- Start the program at optional ADDRESS
step <N> -- Step the program for N instructions
continue -- Continue program execution without stepping
print $N -- Print register N
print $fN -- Print floating point register N
print ADDR -- Print contents of memory at ADDRESS
print_symbols -- Print all global symbols
print_all_regs -- Print all MIPS registers
print_all_regs hex -- Print all MIPS registers in hex
reinitialize -- Clear the memory and registers
breakpoint <ADDR> -- Set a breakpoint at address
delete <ADDR> -- Delete all breakpoints at address
list -- List all breakpoints
dump [ ``FILE'' ] -- Dump binary code to spim.dump or FILE in network byte order
dumpnative [ ``FILE'' ] -- Dump binary code to spim.dump or FILE in host byte order
. -- Rest of line is assembly instruction to put in memory
<cr> -- Newline reexecutes previous command
? -- Print this message

Most commands can be abbreviated to their unique prefix
e.g., ex(it), re(ad), l(oad), ru(n), s(tep), p(rint)

(spim)
```

Listing A.1 The SPIM prompt and help screen.

- The `reinitialize` command is roughly the equivalent to resetting the processor in the sense that it clears the state (including memory and register content).
- The `read` and `load` commands both load a named assembly language program into the simulator ready for execution. As the program is loaded, SPIM might reject it if errors are encountered during the parsing process (for example a syntax error such as a missing comma).
- The `run` command executes the program to completion starting from the entry point defined by the global symbol `__start`; by default, the exception handler includes code which jumps to the label `main` from this entry point. Once run, the program either exits cleanly using an `exit` system call, encounters a non-instruction and crashes, or never terminates.
- In contrast to `run`, the `step` command executes a specific number of instructions and then returns control to the user. The optional argument N specifies the number of instructions to execute. The default is $N = 1$, which is commonly called **single stepping** the program, and allows the inspection of processor state after each instruction is completed to ensure the expected behaviour has resulted.

- A **breakpoint** is a sort of marker one can set in the program; when execution reaches the breakpoint, SPIM returns control to the user rather then continuing with execution. Use of breakpoints offers a compromise between `run` and `step`, for example one might want to execute the program until a particular loop completes, which rules out `run` which would execute until the whole program completes, but is better than using `step` which would demand you know how many instructions the loop will take. The `breakpoint` instruction allows the user to add a breakpoint to the current list. The optional argument allows one to associate a breakpoint with a particular address; the default is to set the breakpoint to the current value of *PC*.
- The `delete` command allows the removal of the breakpoint associated with a particular address. The `list` command prints a list of all current breakpoints. Finally, the `continue` command allows one to continue execution after a breakpoint is encountered; roughly speaking `continue` instructs SPIM to execute the program in a similar way to `run` except starting from where execution broke off instead of from the start.
- Finally, SPIM offers a suite of commands which allow the user to inspect the processor state. The `print` command prints register and memory content. If you supply a register name such as `$1` as the argument, SPIM prints the value held in that register. If you supply a number (in either decimal or hexadecimal form) as the argument, SPIM prints the value held in memory at that address. The `print_all_regs` command offer a short cut way to inspect the whole register file in one go. The `print_symbols` command prints all globally defined symbols and their addresses. This can be useful in locating the physical addresses of labels you referred to abstractly in your program.

Perhaps most importantly from a labour saving point of view, pressing return will repeat the previous command. SPIM interprets this in an intelligent manner so that, for example, using the command `print 0x10010000` to inspect some value in memory and then pressing return executes the command `print 0x10010004` to inspect the subsequent address. Likewise, using the command `print $1` to inspect some value in a register and then pressing return executes `print $2` rather than `print $1` again.

A.4 Example Program Execution

To demonstrate most of the features offered by SPIM, consider the execution of bubble-sort as implemented in Chapter 10 and reproduced here for completeness in Listing A.2.

Listing A.3 demonstrates initial, exploratory use of SPIM. It starts by loading the bubble-sort code using a `load` command; there are no errors in the program so SPIM completes this task silently. We can now inspect the global symbols and their location in memory using `print_symbols`. The noteworthy part of the output is that the symbols defined in our program are present: `a` has been placed by

```
1  .globl A
2  .globl n
3  .globl main
4
5  .data
6
7  A:     .word   2
8         .word   6
9         .word   4
10        .word   3
11        .word   1
12        .word   7
13        .word   5
14        .word   8
15
16 n:     .word   8
17
18 .text
19
20 main: lw      $3,  n($0)  # load n
21       addi    $4,  $3, -1 # set i to n - 1
22 l0:   blez    $4,  l4     # goto l4 if i <= 0
23       add     $2,  $0, $0 # set f to 0
24       add     $5,  $0, $0 # set j to 0
25 l1:   sub     $6,  $4, $5
26       blez    $6,  l3     # goto l3 if i <= j
27       addi    $6,  $0,  4
28       mul     $6,  $6, $5
29       addi    $7,  $6,  4
30       lw      $8,  A($6)  # load A[j  ]
31       lw      $9,  A($7)  # load A[j+1]
32       sub     $10, $8, $9
33       blez    $10, l2     # goto l2 if a[j] <= a[j+1]
34       sw      $9,  A($6)  # store A[j+1] (swap)
35       sw      $8,  A($7)  # store A[j  ] (swap)
36       addi    $2,  $0,  1 # set f to 1
37 l2:   addi    $5,  $5,  1 # set j to j + 1
38       j       l1          # goto l1
39 l3:   blez    $2,  l4
40       addi    $4,  $4, -1 # set i to i - 1
41       j       l0          # goto l0
42 l4:   addi    $2,  $0, 10
43       syscall             # exit
```

Listing A.2 MIPS32 implementation of bubble sort.

SPIM at $10010000_{(16)}$, `n` at $10010020_{(16)}$ and `main` at $00400024_{(16)}$. Rather than worry about their values for the moment, we issue a `print_all_regs` command to inspect the state of the register file (this is truncated to include only the general-purpose register file); essentially the simulator is in the reset state with all registers zeroed. Notice that before the general-purpose registers, SPIM prints a set of special-purpose and control registers. These include *PC*, *LO* and *HI* and registers such as *EPC* which are associated with the management of exceptions and logically are located within coprocessor zero.

We can force SPIM to start executing our program one instruction at a time using `step` commands; where a blank command is shown the user has pressed return. Each line displays the address from which the instruction was loaded, the encoded instruction and the assembly language equivalent (including the comment

```
(spim) load "bubble.asm"
(spim) print_symbols
g       A at 0x10010000
g       n at 0x10010020
g       __eoth at 0x00400024
g       __start at 0x00400000
g       main at 0x00400024
(spim) print_all_regs
 PC      = 00000000   EPC     = 00000000   Cause   = 00000000   BadVAddr= 00000000
 Status  = 3000ff10   HI      = 00000000   LO      = 00000000
                                 General Registers
R0  (r0) = 0          R8  (t0) = 0          R16 (s0) = 0          R24 (t8) = 0
R1  (at) = 0          R9  (t1) = 0          R17 (s1) = 0          R25 (t9) = 0
R2  (v0) = 0          R10 (t2) = 0          R18 (s2) = 0          R26 (k0) = 0
R3  (v1) = 0          R11 (t3) = 0          R19 (s3) = 0          R27 (k1) = 0
R4  (a0) = 0          R12 (t4) = 0          R20 (s4) = 0          R28 (gp) = 268468224
R5  (a1) = 0          R13 (t5) = 0          R21 (s5) = 0
R29 (sp) = 2147479548
R6  (a2) = 0          R14 (t6) = 0          R22 (s6) = 0          R30 (s8) = 0
R7  (a3) = 0          R15 (t7) = 0          R23 (s7) = 0          R31 (ra) = 0
...

(spim) step
[0x00400000]    0x8fa40000  lw $4, 0($29)
(spim)
[0x00400004]    0x27a50004  addiu $5, $29, 4
(spim)
[0x00400008]    0x24a60004  addiu $6, $5, 4
(spim)
[0x0040000c]    0x00041080  sll $2, $4, 2
(spim)
[0x00400010]    0x00c23021  addu $6, $6, $2
(spim)
[0x00400014]    0x0c100009  jal 0x00400024 [main]
(spim)
[0x00400024]    0x3c011001  lui $1, 4097
(spim)
[0x00400028]    0x8c230020  lw $3, 32($1)
(spim) print $3
Reg 3 = 0x00000008 (8)
(spim) step
[0x0040002c]    0x2064ffff  addi $4, $3, -1
(spim) print $4
Reg 4 = 0x00000007 (7)
(spim)
```

Listing A.3 Example use of the SPIM simulator.

from the original source code). The first six instructions look unfamiliar, they are not part of our program. Instead, they are part of the default exception handler which includes code located at the SPIM entry point `__start`: SPIM started execution at `__start` and these six instructions are implementing the call to our program located at `main`. One can think of this as the equivalent to when one executes a compiled C program which needs calls the `main` function. The sixth instruction performs the actual call which we can see since the target address is $00400024_{(16)}$, which matches what we saw in the printed symbols, and SPIM is even kind enough to point out that this address matches the `main` symbol. The next instructions come from our program. For example, the first is `lw $3, n($0)`

```
(spim) load "bubble.asm"
(spim) print A
Data seg @ 0x10010000 (268500992) = 0x00000002 (2)
(spim)
Data seg @ 0x10010004 (268500996) = 0x00000006 (6)
(spim)
Data seg @ 0x10010008 (268501000) = 0x00000004 (4)
(spim)
Data seg @ 0x1001000c (268501004) = 0x00000003 (3)
(spim)
Data seg @ 0x10010010 (268501008) = 0x00000001 (1)
(spim)
Data seg @ 0x10010014 (268501012) = 0x00000007 (7)
(spim)
Data seg @ 0x10010018 (268501016) = 0x00000005 (5)
(spim)
Data seg @ 0x1001001c (268501020) = 0x00000008 (8)
(spim) run
(spim) print A
Data seg @ 0x10010000 (268500992) = 0x00000001 (1)
(spim)
Data seg @ 0x10010004 (268500996) = 0x00000002 (2)
(spim)
Data seg @ 0x10010008 (268501000) = 0x00000003 (3)
(spim)
Data seg @ 0x1001000c (268501004) = 0x00000004 (4)
(spim)
Data seg @ 0x10010010 (268501008) = 0x00000005 (5)
(spim)
Data seg @ 0x10010014 (268501012) = 0x00000006 (6)
(spim)
Data seg @ 0x10010018 (268501016) = 0x00000007 (7)
(spim)
Data seg @ 0x1001001c (268501020) = 0x00000008 (8)
```

Listing A.4 Example use of the SPIM simulator.

intended to load the value of n into `$3`. SPIM expands this into two instructions which are actually executed. We then inspect `$3` using a `print` command to find that the register holds the expected result $00000008_{(16)} = 8_{(10)}$. Executing the next instruction `addi $4, $3, -1` initialises the loop counter `i` by subtracting one from n. Inspecting the register `$4` using `print` again shows the expected result $00000007_{(16)} = 7_{(10)}$.

Lets now inspect the contents of memory. In Listing A.4 we again load the bubble-sort program and then instruct SPIM to print the memory address `a`; SPIM allows us to refer to the address by name. Repeating the command shows the content of the unsorted array. To sort the array, we need to execute the program which is accomplished by issuing a `run` command. The program runs to completion and exits cleanly by using the correct `syscall` whereby control returns to us as the user. To verify that execution was a success, i.e., that array really is sorted, we again print the content.

Finally, lets examine the use of breakpoints to control execution. By telling SPIM to print the program code starting at `main`, we can identify the `blez` instruction at address $00400030_{(16)}$ as relating to the outer loop of bubble-sort: it controls whether or not the loop counter i is greater than zero. As such, it represents a good choice of

```
(spim) load "bubble.asm"
(spim) print main
[0x00400024]    0x3c011001  lui $1, 4097
(spim)
[0x00400028]    0x8c230020  lw $3, 32($1)
(spim)
[0x0040002c]    0x2064ffff  addi $4, $3, -1
(spim)
[0x00400030]    0x1880001c  blez $4 112 [l4-0x00400030]
(spim) breakpoint 0x00400030
(spim) run
Breakpoint encountered at 0x00400030
(spim) print $4
Reg 4 = 0x00000007 (7)
(spim) continue
Breakpoint encountered at 0x00400030
(spim) print $4
Reg 4 = 0x00000006 (6)
(spim) list
Breakpoint at 0x00400030
(spim) delete 0x00400030
(spim) continue
(spim) print A
Data seg @ 0x10010000 (268500992) = 0x00000001 (1)
(spim)
Data seg @ 0x10010004 (268500996) = 0x00000002 (2)
(spim)
Data seg @ 0x10010008 (268501000) = 0x00000003 (3)
(spim)
Data seg @ 0x1001000c (268501004) = 0x00000004 (4)
(spim)
Data seg @ 0x10010010 (268501008) = 0x00000005 (5)
(spim)
Data seg @ 0x10010014 (268501012) = 0x00000006 (6)
(spim)
Data seg @ 0x10010018 (268501016) = 0x00000007 (7)
(spim)
Data seg @ 0x1001001c (268501020) = 0x00000008 (8)
(spim)
```

Listing A.5 Example use of the SPIM simulator.

breakpoint if our objective is to inspect things after each loop iteration. Therefore, we set a breakpoint at address $00400030_{(16)}$ using the `breakpoint` command and then set the program running using `run`. Very quickly, SPIM returns control to us with the message that our breakpoint was encountered. Clearly this has happened before the first loop iteration; we can verify this by printing `$4` which holds i and seeing that it has the value $7_{(10)}$. Using the `continue` command, we set SPIM running again from where it previously stopped. It returns control to us with the message that our breakpoint was encountered again. This time, the breakpoint was after the second loop iteration which we again verify by inspecting i. Fed up with this and convinced our loop is executing correctly, we then list the breakpoints and delete the one we previously set. This time, issuing the `continue` command executes the program to completion: it is no longer interrupted because we deleted the breakpoint. to verify this fact, we print the contents of A and find it is sorted.

System Call Name	System Call Number	Arguments	Result
`print_int`	1	`$a0` is integer	
`print_char`	11	`$a0` is char	
`print_string`	4	`$a0` is string	
`read_int`	5		`$v0` is integer
`read_char`	12		`$a0` is char
`read_string`	8	`$a0` is buffer `$a1` is length	
`exit`	10		
`open`	13	`$a0` is name `$a1` is flags `$a2` is mode	`$a0` is descriptor
`read`	14	`$a0` is descriptor `$a1` is buffer `$a2` is length	`$a0` is chars read
`write`	15	`$a0` is descriptor `$a1` is buffer `$a2` is length	`$a0` is chars written
`close`	16	`$a0` is descriptor	

Table A.1 A table listing a sub-set of system calls supported by SPIM.

A.5 Using System Calls

As previously mentioned, SPIM provides a system call interface that enables programs running within the SPIM simulator to interface with the host system. They capture the sorts of operation that an operating system would normally perform on the behalf of a program (because for example it requires privileged access to some resource). Indeed, the form of SPIM system calls is similar to the interface with a conventional operating system: the program initialises arguments to the system call by storing values in prescribed registers, selects the system call it wants by setting a prescribed register, and then executes the `syscall` instruction. This passes control to the operating system (via the exception handler) which services the request and then returns control to the program along with any results which are again held in prescribed registers). We already used this mechanism informally in previous assembly language program: notice that each program concludes with system call number 10 which is used to exit the program.

The system call convention is detailed in Table A.1. Each entry lists the system call name, the system call number or code and the register allocation vis input argument and output results. The convention dictates that the system call number should be placed in register `$v0` before invocation using the `syscall` instruction. Although the table should be fairly self explanatory, consider the example of

```
.data

prompt: .asciiz "enter a number: "
result: .asciiz "the result is: "

.text

main:   la      $a0, prompt   # set argument
        addi    $v0, $0,    4 # select  syscall  4 (print_string)
        syscall               # perform syscall
        addi    $v0, $0,    5 # select  syscall  5 (read_int)
        syscall               # perform syscall
        add     $t0, $0,  $v0 # store result

        la      $a0, prompt   # set argument
        addi    $v0, $0,    4 # select  syscall  4 (print_string)
        syscall               # perform syscall
        addi    $v0, $0,    5 # select  syscall  5 (read_int)
        syscall               # perform syscall
        add     $t1, $0,  $v0 # store result

        add     $t2, $t0, $t1 # add two read numbers together

        la      $a0, result   # set argument
        addi    $v0, $0,    4 # select  syscall  4 (print_string)
        syscall               # perform syscall
        add     $a0, $0,  $t2 # set argument
        addi    $v0, $0,    1 # select  syscall  1 (print_int)
        syscall               # perform syscall

        addi    $v0, $0,   10 # select  syscall 10 (exit)
        syscall               # perform syscall
```

Listing A.6 Several examples of supported SPIM system calls.

```
(spim) load "syscall.asm"
(spim) run
enter a number: 5
enter a number: 7
the result is: 12
(spim)
```

Listing A.7 Example use of the SPIM simulator.

a program where two numbers are read from the user and summed, and the result is printed to the console. Listing A.6 demonstrates such an implementation with the added embellishment that we also print prompts on the console which are strings allocated with the `.asciiz` directive.

The program first invokes system call 4 (named `print_string`) to print a prompt before using system call 5 (named `read_int`) to read an integer value from the user; this is stored in register `$t0`. The same pattern is used to read a further value from the user which is stored in register `$t1`. These values are added together before system call 1 (named `print_int`) is used to display the result. Finally, system call 10 (named `exit`) is used to cleanly exit the program. An example execution within SPIM is shown in Listing A.7.

Appendix B
Example Solutions

Questions from Chapter 1

1

a. $|A| = 3$.
b. $A \cup B = \{1,2,3,4,5\}$.
c. $A \cap B = \{3\}$.
d. $A - B = \{1,2\}$.
e. $\overline{A} = \{4,5,6,7,8\}$.
f. $\{x : 2 \cdot x \in \mathscr{U}\} = \{1,2,3,4\}$.

2

a. $+0$ in sign-magnitude is 00000000, in twos-complement is 00000000.
b. -0 in sign-magnitude is 10000000, in twos-complement is 00000000.
c. $+72$ in sign-magnitude is 01001000, in twos-complement is 01001000.
d. -34 in sign-magnitude is 10100010, in twos-complement is 11011110.
e. -8 in sign-magnitude is 10001000, in twos-complement is 11111000.
f. Trick question, one cannot represent 240 in 8-bit sign-magnitude or twos-complement; the incorrect guess of 11111000 in twos-complement for example is actually -8.

3 The Hamming weight of x, denoted by $H(x)$ say, is the number of bits in the representation which are set to one. A function to compute this can be written as:

```
int H( int x ) {
  int w = 0;

  for( int i = 0; i < 32; i++ ) {
    if( ( x >> i ) & 1 )
      w = w + 1;
  }

  return w;
}
```

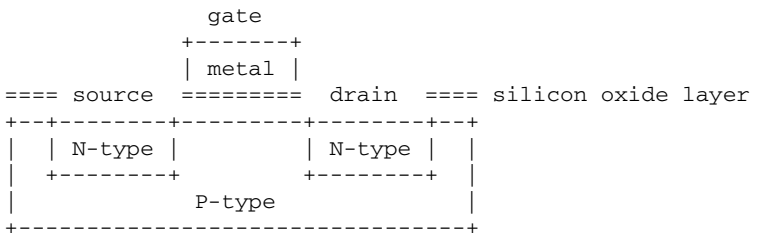
4 Since $z \vee \neg z = 1$ and $z \vee z = z$, we have that

$$\begin{aligned}\neg x \vee \neg y &= (\neg x \wedge (y \vee + \neg y)) \vee (\neg y \wedge (x \vee + \neg x)) \\ &= (\neg x \wedge y) \vee (\neg x \wedge \neg y) \vee (\neg y \wedge x) \vee (\neg y \wedge \neg x) \\ &= (\neg x \wedge y) \vee (\neg y \wedge x) \vee (\neg x \wedge \neg y).\end{aligned}$$

Questions from Chapter 2

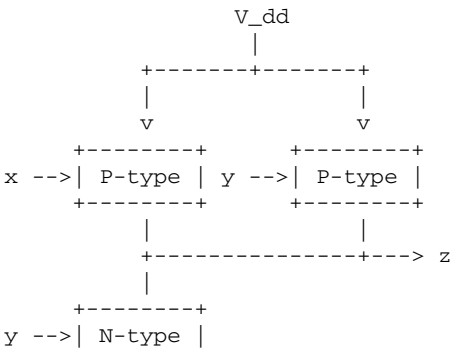
5

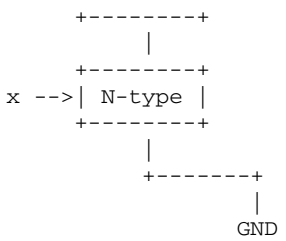
a. MOSFET transistors work by sandwiching together N-type and P-type semiconductor layers. The different types of layer are doped with different substances to create more holes or more electrons. For example, in an N-type MOSFET the layers are constructed as follows



with additional layers of silicon oxide and metal. There are three terminals on the transistor. Roroughly speaking, applying a voltage to the gate creates a channel between the source and drain through which charge can flow. Thus the device acts like a switch: when the gate voltage is high, there is a flow of charge but when it is low there is little flow of charge. A P-type MOSFET swaps the roles of N-type and P-type semiconductor and hence implements the opposite switching behaviour.

b. One can construct an NAND to compute $z = x \overline{\wedge} y$ gate from such transistors as follows





When input x is connected to GND, the bottom transistor will be closed while the top left transistor will be open: the output r will be connected to V_{dd} no matter what input y is connected to. Likewise, when input y is connected to GND the middle transistor will be closed while the top right transistor will be open: the output will be connected to V_{dd} no matter what input x is connected to. However when both x and y are connected to V_{dd} we have that the top two transistors will be closed by the bottom two will be open meaning that the output is connected to GND.

6

a. The most basic interpretation (i.e., not really doing any grouping using Karnaugh maps but just picking out each cell with a 1 in it) generates the following SoP equations

$$e = (\neg a \wedge \neg b \wedge c \wedge \neg d) \vee (a \wedge \neg b \wedge \neg c \wedge \neg d) \vee (\neg a \wedge b \wedge \neg c \wedge d)$$
$$f = (\neg a \wedge \neg b \wedge \neg c \wedge d) \vee (\neg a \wedge b \wedge \neg c \wedge \neg d) \vee (a \wedge \neg b \wedge c \wedge \neg d)$$

b. From the basic SoP equations, we can use the don't care states to eliminate some of the terms to get

$$e = (\neg a \wedge \neg b \wedge c) \vee (a \wedge \neg c \wedge \neg d) + (b \wedge d)$$
$$f = (\neg a \wedge \neg b \wedge d) \vee (b \wedge \neg c \wedge \neg d) + (a \wedge c)$$

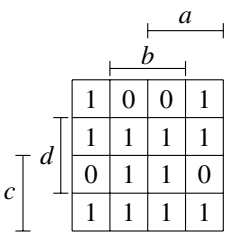
then, we can share both the terms $\neg a \wedge \neg b$ and $\neg c \wedge \neg d$ since they occur in e and f.

7

a. The expression for this circuit can be written as

$$e = (\neg c \wedge \neg b) \vee (b \wedge d) \vee (\neg a \wedge c \wedge \neg d) \vee (a \wedge c \wedge \neg d)$$

which yields the Karnaugh map



and from which we can derive a simplified SoP form for e using the techniques in the associated chapter

$$e = (b \wedge d) \vee (\neg b \wedge \neg c) \vee (c \wedge \neg d)$$

b. The advantages of this expression over the original are that is is simpler, i.e., contains less terms and hence needs less gates for implementation, and shows that the input a is essentially redundant. We have probably also reduced the critical path through the circuit since it is more shallow. The disadvantages are that we still potentially have some glitching due to the differing delays through paths in the circuit, although these existed before as well, and the large propagation delay.
c. The longest sequential path through the circuit goes through a NOT gate, two AND gates and two OR gates; the critical path is thus 90ns long. This time bounds how fast we can used it in a clocked system since the clock period must be at least 90ns. So the fastest we can use it would have the clock period at 90ns which means the clock ticks about 11111111 times a second or at about 11 MHz.

8 Let a head be denoted by 1 and a tail by 0 and assume we have a circuit which produces a random coin flip R every clock cycle.

There are several approaches to solving this problem. Possibly the easiest, but perhaps not the most obvious, is to simply build a shift-register: the register stores the last three inputs, when a new input R is available the register shifts the content along by one which means the oldest input drops off one end and the new input is inserted into the other end. One can then build a simple circuit to test the current state of the shift-register to see if the last three inputs were all heads.

Alternatively, one can take a more heavy-weight approach and formulate the solution as a state machine. Our machine can have seen $0 \ldots 3$ heads in the last 3 coin flips it observed; thus we can represent the current state in 2 bits named S_0 and S_1 and the next state in 2 bits named S'_0 and S'_1. The flag that denotes that we have seen 3 heads in a row is called F.

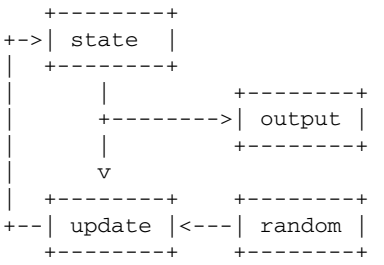
Assuming when we see a head and have already seen 3 heads we can still say we have seen 3 consecutive heads, i.e., the machine is not reset, we can draw the state transition table as:

S_1	S_0	R	S'_1	S'_0	F
0	0	0	0	0	0
0	1	0	0	0	0
1	0	0	0	0	0
1	1	0	0	0	0
0	0	1	0	1	0
0	1	1	1	0	0
1	0	1	1	1	1
1	1	1	1	1	1

From this we need expressions for S'_0 and S'_1 and F which can easily be derived as

$$\begin{aligned} S'_0 &= R \wedge ((\neg S_0) \vee (S_1 \wedge S_0)) \\ S'_1 &= R \wedge ((S_0) \vee (S_1 \wedge \neg S_0)) \\ F &= R \wedge S_1 \end{aligned}$$

Putting this all together, we can build a state machine with the following block diagram:



9

a. There are several approaches to solving this problem. Possibly the easiest, but perhaps not the most obvious, is to simply build a shift-register: the register stores the last three inputs, when a new input is available the register shifts the content along by one which means the oldest input drops off one end and the new input is inserted into the other end. One can then build a simple circuit to test the current state of the shift-register to see if the last three inputs match what is required..
Alternatively, one can take a more heavy-weight approach and formulate the solution as a state machine. First we need to decide on an encoding for our state; when searching though the input we can have matched zero through three correct tokens we denote this by the integer S stored in two bits using S_1 and S_0 as the most-significant and least-significant respectively. We also need an encoding of the actual input tokens I which are being passed to the matching circuit. Arbitrarily we might select $A = 0$, $C = 1$, $G = 2$ and $T = 3$ although other encodings are valid and might actually simplify things; we use I_1 and I_0 to denote the most and least-significant bits of the input token I. From this we can now create a table describing the mapping between current state S and input I to next state S' which

can be roughly written as

S_1	S_0	I_1	I_0	S'_1	S'_0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

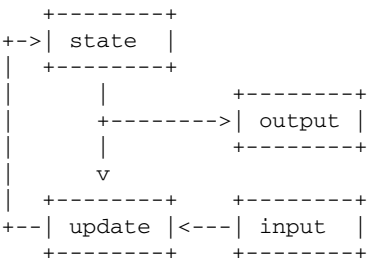
Thus, if we are in state $(S_1, S_0) = (0,0) = 0$ and see an A input, we move to state $(S'_1, S'_0) = (0,1) = 1$ otherwise we stay in state $(S'_1, S'_0) = (0,0) = 0$. Now we can define the transition function from current state to next state as

$$S_0 = (\neg S_1 \wedge \neg S_0 \wedge \neg I_1 \wedge \neg I_0) \vee (S_1 \wedge \neg S_0 \wedge I_1 \wedge I_0) \vee (S_1 \wedge S_0 \wedge \neg I_1 \wedge \neg I_0)$$
$$S_1 = (\neg S_1 \wedge S_0 \wedge \neg I_1 \wedge I_0) \vee (S_1 \wedge \neg S_0 \wedge I_1 \wedge I_0)$$

with simplifications as appropriate. Finally, the output flag F will be set only according to

$$F = S_1 \wedge S_0$$

to signal when we have matched three characters. Putting this all together, we can build a state machine with the following block diagram:



b. Making a general purpose matching circuit will probably use less logic than having three separate circuits; this will reduce the space required. As an extension one might consider implementing the transition and output functions as a look-up table instead of hardwiring them; this will mean the circuit could be used to

match any sequence providing the tables were correctly initialised. Introducing a more complex circuit design could have the disadvantage of increasing the critical path (the longest sequential path though the entire circuit). If the critical path is longer, the design will have to be clocked slower and hence will not perform the matching function as quickly.

10 Define $\overline{\wedge}$ as the NAND operation with the truth table:

x	y	$x \overline{\wedge} y$
0	0	1
0	1	1
1	0	1
1	1	0

Using NAND, we can implement NOT, AND and OR as follows:

$$\begin{aligned} \neg x &= x \overline{\wedge} x \\ x \wedge y &= (x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y) \\ x \vee y &= (x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y) \end{aligned}$$

To prove this works, we can construct truth tables for the expressions and compare the results with what we would expect; for NOT we have:

x	$x \overline{\wedge} x$	$\neg x$
0	1	1
1	1	0

while for AND we have:

x	y	$x \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$x \wedge y$
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

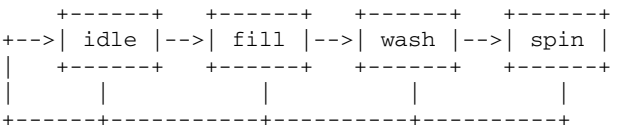
and finally for OR we have:

x	y	$x \overline{\wedge} x$	$y \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$x \vee y$
0	0	1	1	0	0
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	1	1

such that it should be clear all three are correct.

11

a. A basic diagram should show the four states and transitions between them which relate the movement from one to the other as a result of the washing cycle, and movement as a result of input from the buttons; for example a (very) basic diagram would be:

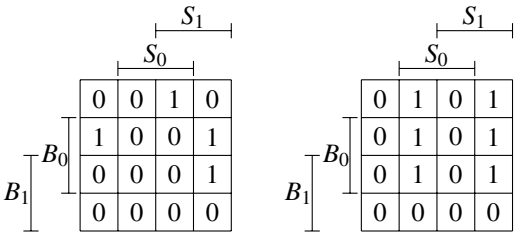


b. Since there are four states, we can encode them using one two bits; we assign the following encoding *idle* = 00, *fill* = 01, *wash* = 10 and *spin* = 11. We use S_1 and S_0 to represent the current state, and N_1 and N_0 to represent the next state; B_1 and B_0 are the input buttons. Using this notation, we can construct the following state transition table which encodes the state machine diagram:

B_1	B_0	S_1	S_0	N_1	N_0
0	0	0	0	0	0
0	1	0	0	0	1
1	0	0	0	0	0
1	1	0	0	0	0
0	0	0	1	1	0
0	1	0	1	1	0
1	0	0	1	0	0
1	1	0	1	1	0
0	0	1	0	1	1
0	1	1	0	1	1
1	0	1	0	0	0
1	1	1	0	1	1
0	0	1	1	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	1	0	0

so that if, for example, the machine is in the *wash* state (i.e., $S_1 = 1$ and $S_0 = 0$) and no buttons are pressed then the next state is *spin* (i.e., $N_1 = 1$ and $N_0 = 1$); however if button B_1 is pressed to cancel the cycle, the next state is *idle* (i.e., $N_1 = 0$ and $N_0 = 0$).

c. From the state transition table, we can easily extract the two Karnaugh maps:



Basic expressions can be extracted from the tables as follows:

$$N_0 = (S_1 \wedge \neg S_0 \wedge B_0) \vee (S_1 \wedge \neg S_0 \wedge \neg B_1) \vee (\neg S_0 \wedge B_0 \wedge \neg B_1)$$
$$N_1 = (\neg S_1 \wedge S_0 \wedge B_0) \vee (\neg S_1 \wedge S_0 \wedge \neg B_1) \vee (S_1 \wedge \neg S_0 \wedge B_0) \vee (S_1 \wedge \neg S_0 \wedge \neg B_1)$$

and through sharing, i.e., by computing

$$t_0 = \neg B_1$$
$$t_1 = \neg B_0$$
$$t_2 = \neg S_1$$
$$t_3 = \neg S_0$$
$$t_4 = t_2 \wedge S_0$$
$$t_5 = t_3 \wedge S_1$$

we can simplify these to

$$N_0 = (t_5 \wedge B_0) + (t_5 \wedge t_0) + (t_3 \wedge B_0 \wedge t_0)$$
$$N_1 = (t_4 \wedge B_0) + (t_4 \wedge t_0) + (t_5 \wedge B_0) + (t_5 \wedge t_0)$$

Questions from Chapter 3

12

```
a. wire        [ 7:0] a;
b. wire        [ 0:4] b;
c. reg         [31:0] c;
d. reg signed  [15:0] d;
e. reg         [ 7:0] e[0:1023];
f. genvar             f;
```

13

```
a. c = 2'b01
b. c = 2'b11
c. d = 4'b010X
d. d = 4'b10XX
e. d = 4'b1101
f. d = 4'b0111
```

g. `c = 2'bXX`
h. `c = 2'b11`
i. `e = 1'b0`
j. `e = 1'b1`

14

a. One potential problem is the if `p` and `q` can change at any time, and hence trigger execution of the processes at any time, the two might change at the exact *same* time. In this case, it is not clear which values `x` and `y` will be assigned to. Maybe the top assignment to `x` beats the bottom one, but the bottom assignment to `y` beats the top one. Any combination is possible; since it is not clear which will occur, it is possible that `x` and `y` do not get assigned to the values one would expect.
As an attempt at a solution, we can try to exert some control over which block of assignments is executed first. For example, we might try to place a guard around the assignments so that a block is only executed if the signal trigger for the other block is low:

```
always @ ( posedge p )
begin
  if( !q )
  begin
    x <= a;
    y <= b;
  end
end

always @ ( posedge q )
begin
  if( !p )
  begin
    x <= b;
    y <= a;
  end
end
```

An alternative might be to combine the two blocks into one:

```
always @ ( posedge p, posedge q )
begin
  if     ( p )
  begin
    x <= a;
    y <= b;
  else if( q )
  begin
    x <= b;
    y <= a;
  end
end
```

since now the process is at least deterministic: if `p` is high then the first block executes, if `q` is high then the second block executes and if both are high we execute the first block as the default.

b. The problem with this is that the `state` signal is not initialised; to start with it could be any value which might either result in the state machine operating in the wrong sequence or, since `case` is used and not `casex` or `casez`, none of the cases being matched at all. A slightly more minor issue is that we have to assume that no other process assigns to `state`. For example, if another process sets `state` to 3 the state machine process will malfunction.
The best way to rectify this problem is by introducing a reset signal called `rst` and initialising the state variable whenever it is raised:

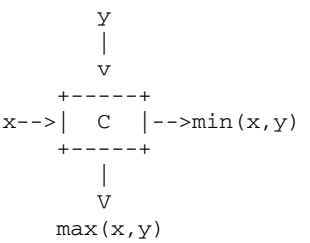
```
always @ ( posedge clk, posedge rst )
begin
  if( rst )
    state = 0;
  else
    case( state )
      0 : begin
            do_0;
            state = 1;
          end
      1 : begin
            do_1;
            state = 2;
          end
      2 : begin
            do_2;
            state = 0;
          end
end
```

15 This is a bit of a vague question; it does not mention styles of Verilog and so you can assume any valid style is okay. With this in mind, a rough solution might look something like this:

```
module DDD( clk, rst, in, out );

  input  wire      clk;
  input  wire      rst;

  input  wire      bit;
  output wire      out;

         reg [3:0] t;

  assign out = ( t >= 0 ) && ( t <= 9 );

  always @ ( posedge clk )
  begin
    if( rst )
```

```
      t = 0;
    else
      t = { bit, t[3:1] };
  end

endmodule
```

16 Essentially we want a design that looks, at a high-level, like this:



and thus need some sort of comparison inside; we already know how to design a less than comparison which is good enough. Thus, the Verilog module could look something like this:

```
module C( x, y, min, max );

  parameter n = 8;

  input  wire [n-1:0]   x;
  input  wire [n-1:0]   y;
  output wire [n-1:0] min;
  output wire [n-1:0] max;

         wire [n-1:0] w0 = ~(  x ^ y );
         wire [n-1:0] w1 =  ( ~x & y );
         wire [n-1:0] w2;

  assign w2[0] = w1[  0];

  genvar i;
  generate
  begin
    for( i = 1; i < n; i = i + 1 )
    begin:gen_lth
      assign w2[i] = w1[i] | ( w0[i] & w2[i-1] );
    end
  end
  endgenerate

  assign min = ( w2[n-1] ) ? x : y;
  assign max = ( w2[n-1] ) ? y : x;

endmodule
```

17 In principle, the design we want is fairly simply: we keep track of the 8-bit state in a register, and then on each clock edge drive the 0-th bit onto an output wire and then shift the register (producing a new input bit as required). Thus, the Verilog module could look something like this:

```
module LFSR( clk, rst, out, seed );

  input  wire       clk;
  input  wire       rst;

  output reg        out;
  input  wire [7:0] seed;
         reg  [7:0] lfsr;

  always @ ( posedge clk )
  begin
    if( rst )
      out  =    0;
      lfsr = seed;
    else
      out  = lfsr[0];
      lfsr = { lfsr[3]^lfsr[5]^lfsr[7], lfsr[7:1] };
  end

endmodule
```

Questions from Chapter 4

18 In a Harvard architecture, since the data and instruction memories are separated from each other they can:

a. Be built in fundamentally different ways. For example one might remove logic associated with store operations from the instruction memory, or allow the instruction memory to have a different resolution (i.e., hold different sized values). This permits the designer to capitalise on differences use cases and satisfy different metrics for quality (e.g., size, power consumption or performance).
b. Be accessed in parallel. For example, in order to reduce the time taken to complete a given fetch-decode-execution cycle, the processor might attempt to fetch the next instruction at the same time as executing the current one. The fact that there are two memories (and hence two memory interfaces) allows this, whereas in a "pure" stored program architecture one has to perform the accesses in sequence.
c. Prevent some problems associated with self-modifying code. Examples include security threats which result from the ability for an attacker to alter the program at run-time in a way the programmer did not intend.

19 A very simply implementation of such an emulator is as follows:

```
int MEM[] = { ... };

int main( int argc, char* argv[] )
{
  int IR = 0, PC = 0, A = 0;

  do
  {
    IR = MEM[ PC ];
    PC = PC + 1;

    int t0 = IR / 100000;
    int t1 = IR % 100000;

    switch( t0 )
    {
      case 0 : A = t1;
               break;

      case 1 : MEM[ t1 ] = A;
               break;

      case 2 : A = MEM[ t1 ];
               break;

      case 3 : A = A + MEM[ t1 ];
               break;

      case 4 : A = A - MEM[ t1 ];
               break;

      case 5 : break;

      case 6 : PC = t1;
               break;

      case 7 : if( A == 0 )
                 PC = t1;
               break;

      case 8 : if( A != 0 )
                 PC = t1;
               break;
    }
  }
  while( t0 != 5 );
}
```

The idea is to represent the processor state using the variables `IR`, `A` and `PC` and to have an array called `MEM` that represents the memory content. When the emulator is executed, the first assignments "reset" the processor by initialising `IR`, `A` and `PC`. It subsequently enters a loop which only terminates when a *HALT* instruction is encountered. Within the loop, the fetch-decode-execute cycle is emulated:

a. First the current instruction is loaded from `MEM` using the address in `PC` then `PC` is incremented to point at the next instruction.
b. The instruction in `IR` is "decoded" into an opcode and operand that are held in `t0` and `t1`.
c. Finally the instruction is executed by checking `t0` and carrying out an operation which matches the semantics of the opcode; for example if `t0` is 1 we want to perform a store of the accumulator into memory so therefore execute the statement `MEM[ t1 ] = A` since `t1` gives the address we want to store into.

20 As an example, imagine $n = 8$ and $m = 13$: the array X has eight elements which occupy $MEM[13 \ldots 20]$. Once we know n and m, the easiest way to to compute the sum is to use $n-1$ addition instructions to accumulate the result into A and then store it into memory (for example at $MEM[12]$):

Address	Content		Meaning
0	200013	$\mapsto$	$A \leftarrow MEM[13]$
1	300014	$\mapsto$	$A \leftarrow A + MEM[14]$
2	300015	$\mapsto$	$A \leftarrow A + MEM[15]$
3	300016	$\mapsto$	$A \leftarrow A + MEM[16]$
4	300017	$\mapsto$	$A \leftarrow A + MEM[17]$
5	300018	$\mapsto$	$A \leftarrow A + MEM[18]$
6	300019	$\mapsto$	$A \leftarrow A + MEM[19]$
7	300020	$\mapsto$	$A \leftarrow A + MEM[20]$
8	100012	$\mapsto$	$MEM[12] \leftarrow A$
9	500000	$\mapsto$	$HALT$
10			
11			
12	0		
13	$X[0]$		
14	$X[1]$		
15	$X[2]$		
16	$X[3]$		
17	$X[4]$		
18	$X[5]$		
19	$X[6]$		
20	$X[7]$		

Of course, this is a somewhat trivial solution in the sense that we have used an unrolled loop. Adopting a different approach and computing the sum via iteration over some loop body produces the following:

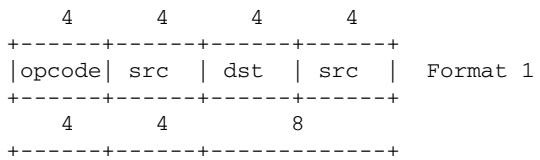
Address	Content		Meaning
0	200013	$\mapsto$	$A \leftarrow MEM[13]$
1	300012	$\mapsto$	$A \leftarrow A + MEM[12]$
2	100012	$\mapsto$	$MEM[12] \leftarrow A$
3	200000	$\mapsto$	$A \leftarrow MEM[0]$
4	300010	$\mapsto$	$A \leftarrow A + MEM[10]$
5	100000	$\mapsto$	$MEM[0] \leftarrow A$
6	200000	$\mapsto$	$A \leftarrow MEM[0]$
7	400011	$\mapsto$	$A \leftarrow A - MEM[11]$
8	800000	$\mapsto$	$ifA \neq 0, PC \leftarrow 0$
9	500000	$\mapsto$	$HALT$
10	1		
11	200021		
12	0		
13	$X[0]$		
14	$X[1]$		
15	$X[2]$		
16	$X[3]$		
17	$X[4]$		
18	$X[5]$		
19	$X[6]$		
20	$X[7]$		

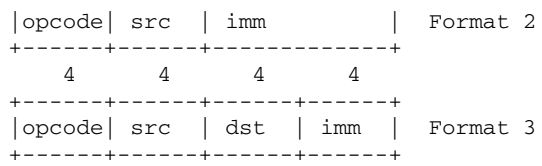
Instructions #0 to #2 update the current value of s by loading and accumulating the i-th element of X. Instructions #3 to #5 self-modify instruction #0 which loads the i-th element: this has the effect of changing it so that in the next iteration, it will loads the $(i+1)$-th element (and so on). Instructions #6 to #8 control the loop by loading a semi-artificial loop bound from $MEM[11]$ and comparing it with $MEM[0]$: if the result is negative then further iterations are required. Finally instruction #9 halts the program when enough iterations have been completed.

Questions from Chapter 5

21

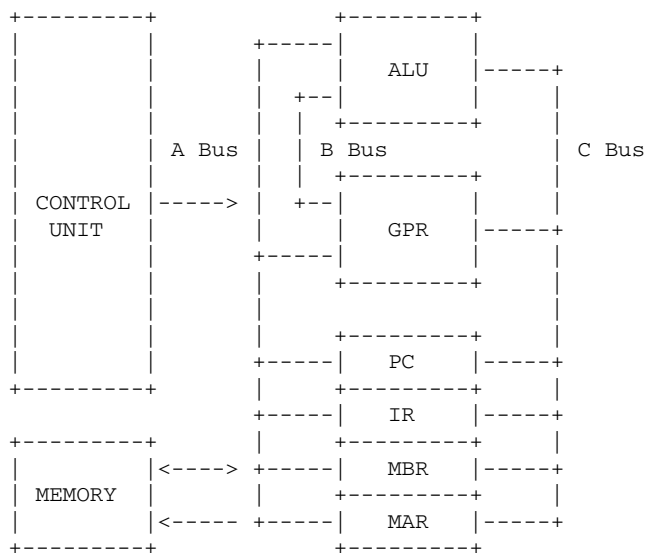
a. We have a total of 16 registers to address; each address therefore requires 4 bits. There are 16 instructions in total; the opcode therefore requires 4 bits. To accommodate this, we define three formats:





Format 1 is used for ALU and register based memory instructions; format 2 is for branches; format 3 is for immediate based memory instructions. Having three formats is good, fewer formats means a more complex processor; having fixed width 16-bit formats, variable length encodings mean a more complex processor. We have also organised the instruction fields so they are aligned, for example the opcode is always the top 4 bits and the next 4 bits is always a source register operand.

b. A reasonable diagram might resemble:



where

- *ALU* is the Arithmetic and Logic Unit.
- *GPR* is the General-Purpose Register file.
- *PC* is the Program Counter.
- *IR* is the Instruction Register.
- *MBR* is the Memory Buffer Register.
- *MAR* is the Memory Address Register.

A register file holds general-purpose, addressable registers which store data operated on by instructions. The program counter stores the address of the next instruction to be executed; the instruction register holds the instruction currently

being executed. The memory data and address registers operate the interface to memory: the processor sends and receives data in the memory data register; it offers the address for loads and stores in the memory address register. The control unit operates the data-path signals to actually make the processor run. The arithmetic and logic unit (ALU) performs computation on source values to produce results so that meaningful processing can be carried out. The components are connected with several buses, most notably between the ALU and the registers; between memory and the processor. Smaller control signals connect the control unit with all the components so it can operate them.

c. The fetch stage loads the instruction from memory. The processor first copies the program counter to the memory address and then starts the access cycle to retrieve the instruction. The program counter is then incremented to point at the next instruction. The decode stage decides what the fetched instruction means in terms of what the processor must do to execute it. Typically this will include extracting and sign-extending any immediate values and fetching source values from the register file. The execute stage performs the actual computation of the instruction, in this case the processor sends the source values through the ALU which adds them together and stores the result into a latch register before writing them into the register file.

d. Hardwiring the control logic means to explicitly encode all the conditions the processor must make when controlling the data-path signals into dedicated logic. This is easier from a design perspective but has the drawback that once implemented, is much harder to change. A micro-code based design takes the opposite approach and stores the information the control unit needs in the form of a very simple program that controls the data-path signals. This program is much lower-level than that which is exposed to the programmer and must be stored in some dedicated on-chip memory. The advantage is that the micro-code can change, and hence the processor operation can be altered, without changing the actual data-path logic.

22 Immediate addresses are constant values held in the instruction; one commonly uses such addresses in C statements such as `a=b+10;` where the instruction would include the register addresses for `a` and `b` and the constant 10. Direct addressing is where we pass an address to the memory and we get back the value at exactly that address, i.e., we give address x and get back $MEM[x]$. This might be the case when using any variable. For example in the C statement `a=b+10;` if `a` or `b` are stored in memory rather than registers we need to load them before using them; if we know their exact address this might be achieved using direct addressing. Indirect addressing is a two-stage process; we pass the memory address x and get back $MEM[MEM[x]]$ so the access is like a pointer dereference. That is, if we want to dereference a pointer `p` which is stored at a particular address x in memory, we first load `p` with $MEM[x]$ and then dereference the pointer by looking in memory where it points so end up with $MEM[MEM[x]]$. In C this might look like the expression `*p`. Finally, indexed addressing means that the value we get back is located at an address formed from the sum of a base and an offset, that is we get $MEM[x]$ for an x formed by evaluating base + offset. This sort of operation is commonly used for C

array accesses, for example the expression `A[10]` is evaluated by taking the base, i.e., the address of `A`, and adding the offset 10.

23

a. The basic idea behind predicated execution is that each instruction as a simple predicate function associated with it. When the instruction is fetched and is ready to be executed, the predicate function is evaluated: if it evaluates to true, the function is executed, otherwise it is discarded.
Assume the symbols t, x, i and n are allocated the registers $GPR[1]$, $GPR[2]$, $GPR[3]$ and $GPR[4]$. The ARM scheme for predicated execution represents a somewhat standard approach: execution of each arithmetic instruction updates a status register which contains flags that detail whether the result generated a carry (C), generated an overflow (O), was negative (N) or was zero (Z). Each instruction can include a predicate which allows execution or non-execution based on these flags. As such, we could translate the fragment into the assembly language style operations:

$$
\begin{array}{ll}
 & \ldots \\
 & GPR[1] \leftarrow 0 \\
 & GPR[3] \leftarrow 0 \\
loop: & \text{if } GPR[3] \geq GPR[4], PC \leftarrow exit \\
 & GPR[4] \leftarrow GPR[2] \gg GPR[3] \\
 & GPR[4] \leftarrow GPR[4] \wedge 1 \\
 & \text{if } Z = 0, GPR[1] \leftarrow GPR[1] + 1 \\
 & GPR[3] \leftarrow GPR[3] + 1 \\
 & PC \leftarrow loop \\
exit: & \ldots
\end{array}
$$

Two conventional branches have been used to implement the loop: the reasoning is that they transfer control-flow over a (fairly) large number of dissimilar instructions which might themselves update the status register. Furthermore, the branches can be fairly accurately predicted using even static techniques so one would not expect too much of a problem in terms of their preventing effective speculative execution. On the other hand, the branch over the single instruction update of t is essentially random (for arbitrary x) so branch prediction is difficult and expensive in relation to the instruction which the change of control-flow is skipping over. As a result, predicated execution is used to skip the instruction based on the Z flag which would have been set via calculation of the conditional expression.

b. Roughly speaking, code density refers to the number of instructions (or amount of work done) per-unit of memory. That is, if we have 100 instructions that fit into 100 bytes of memory the code density is the ratio of the two, i.e., $\frac{100}{100} = 1$. If we fit more instructions into our 100 bytes, say 150, then the code density increases $\frac{150}{100} = 1.5$. Good code density is viewed as an advantage because it typically reduces the code footprint for programs, i.e., the amount of memory

they consume, or means that instructions are smaller, and hence easier or faster to retrieve.

There are plenty of ways to improve code density. One very involved example is to redesign the ISA so that each instruction can accomplish more; this is roughly the approach of CISC processors. In this way, since each instruction performs more useful work, there need be less instructions in total. Another way is to write better programs, or use a better compiler, that is able to use less instructions to achieve the same high-level goal. Clearly there is a limit to how successful this can be, but clever programming can often drive the code footprint underneath some fixed barrier that prevents execution; perhaps the processor has 100kB of memory for example and the program needs 101kB before one optimises it using clever programming.

The concept of variable length instruction potentially increases code density at the cost of increased complexity within the instruction decoding stage. By allowing each instruction to be as long or short as required, instructions that might normally consume a lot of space because they are forced to be the same length as others, can be smaller and hence save space.

24 First we look at the types of instruction required:

a. 0 registers, 1 immediate of 7 bits to store $0 \ldots 127$.
b. 1 registers, 1 immediate of 4 bits to store $-8 \ldots 7$.
c. 2 registers, 0 immediate
d. 1 registers, 0 immediate
e. 0 registers, 0 immediate

Given register indicies are representing as 2-bit operands, a possible encoding has 5 instruction types:

a. `0xxxxxxx`
b. `10rrxxxx`
c. `11ccrrrr` → `1100rrrr` ... `1110rrrr`
d. `1111ccrr` → `111100rr` ... `111110rr`
e. `111111cc` → `11111100` ... `11111111`

where `c` represents space for opcodes, `x` represents space for an immediate and `r` represents space for register indicies. From this we can see that there are 3 possible 2 operand instructions, 3 possible 1 operand instructions and 4 possible 0 operand instructions.

From this we can start encoding the actual instructions already given:

```
LDA x   → 0xxxxxxx
BEZ R x → 10RRxxxx
MOV R S → 1100RRSS
NOT R   → 111100RR
NEG R   → 111101RR
NOP     → 11111100
```

The additional instructions come from the obvious classes: so far we have no load or store instructions, no add instruction which is quite fundamental, and no comparisons.

25 This is quite a vague question and demands that you make (stated) assumptions about anything that is unclear.

A basic approach is as follows; there are 64 squares on the board so one can encode a board position as a 6-bit integer. To encode a move, we simply use two such board positions, a source s and a target t with the meaning that one moves the piece at board position s to board position t. This is a 12-bit representation which is not too compact but is quite easy to decode.

To be more compact, we need to be more clever. Firstly, note that the robot keeps track of whose turn it is so to identify a piece, in some sense we don't need to encode its colour: this is implicit in whose move it is. Secondly note that each piece cannot move to an arbitrary board position: the movement rules constrain how many places it can end up. For example, a king can only move to the squares directly around it: 8 possible moves in total not 64. Drawing a table for the other pieces we find that

Piece	Moves	$\log_2(\text{Moves})$
King	8	3
Queen	27	5
Bishop	13	4
Knight	8	3
Rook	14	4
Pawn	4	2

That is, for example, a queen can move in 27 possible ways and we can encode these ways as a 5-bit integer. So, gathering the different types together we find that there are First we look at the types of instruction required and find there are four:

a. 1 instruction (move queen) with 5-bit operands.
b. 4 instructions (move bishop or rook) with 4-bit operands.
c. 3 instructions (move king and knight) with 3-bit operands.
d. 8 instructions (move pawn) with 2-bit operands.

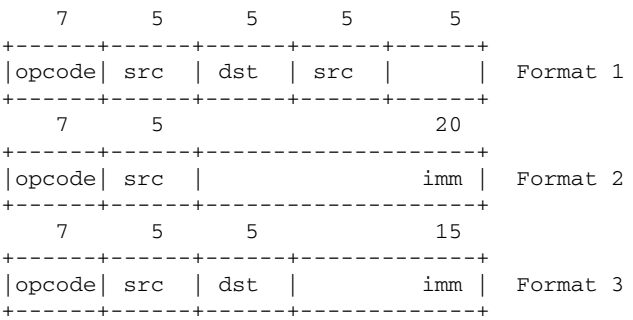
With four types, we can produce an encoding using 8-bit integers:

a. `00cxxxxx` → `000xxxxx` ... `001xxxxx`
b. `01ccxxxx` → `0100xxxx` ... `0111xxxx`
c. `10cccxxx` → `10000xxx` ... `01111xxx`
d. `11ccccxx` → `110000xx` ... `011111xx`

where `c` represents space for opcodes, `x` represents space for an immediate. This is slightly harder to decode but is much more compact. One can probably form a 7-bit encoding by simply enumerating the possible pieces against their possible moves and associating an integer with each combination. However, this could be harder to decode (although this is maybe debatable it one simply has a big look-up table).

26

a. A fixed length encoding is where each instruction is of the same length; this is an advantage in that the decode and fetch system within the processor can be simple and each instruction typically only requires one access to memory. A variable length encoding allows the instructions to be longer or shorter depending on how many operands etc. they require. This makes the processor more complex but can potentially improve the code density of programs.
Following the MIPS32 instruction set architecture, we can define three fixed length instructions formats (which roughly correspond to the MIPS32 R, J and I formats) as follows:



Format 1 is used for ALU and register based memory instructions; format 2 is for branches; format 3 is for immediate based memory instructions. Having three formats is good, fewer formats means a less complex processor; having fixed width 16-bit formats means a less complex processor; having organised the instruction fields so they are aligned, for example the opcode is always the top 4 bits and the next 4 bits is always a source register operand, means a less complex processor. Other advantages include the facility for large immediate values in both branch and ALU instructions; disadvantages include a waste of space in some cases which could be put to better use if more information about the processor requirements was given (for example maybe the space could be used to implement predicate flags for predicated execution).

b. To encode the "add immediate" we would use format 3 from our design; assume that the opcode for this instruction is 3 and that immediate values are held in twos-complement form. Thus, the encoded instructions would look like

$$GPR[4] \leftarrow GPR[8] + 10 \rightarrow \texttt{0000011 01000 00100 0000000000001010}$$
$$GPR[4] \leftarrow GPR[8] - 10 \rightarrow \texttt{0000011 01000 00100 1111111111110110}$$

Notice the subtle sign case of -10 in the sense that we need to encode this as a 15-bit signed value.

c. This function performs a population count on the value x: it counts the number of bits in x which are set to 1. One simple way to help the customer achieve good performance in their application is to support this function in hardware as an ALU operation rather than have them implement it as a C function. We need to expose this operation to the programmer via the instruction set so that,

for example, an instruction of the form `pop $1, $2` would take the value in general-purpose register 2, perform the population count operation and write the result into general-purpose register 1. The customer could then replace every call to the C function with a single, fast machine instruction.

Of course, an appropriate circuit is required within the ALU to perform the population count operation. A clocked circuit to do this would roughly mirror the C implementation, but this would take 32 clock cycles to complete which is still quite slow. A better approach would be to build a combinatorial circuit; the goal with a such a circuit would be to minimise the critical path so that the operation can complete in one cycle. One approach would be to construct a binary adder tree which starts (at the leaves of the tree) by adding two bits of the input together: this gives the number of bits set in that region. Roughly, the circuit computes:

$$((x_0 + x_1) + (x_2 + x_3)) \cdots ((x_{28} + x_{29}) + (x_{30} + x_{31}))$$

so that the end result is the number of bits set to 1. This circuit would have a depth of $\log_2(32) = 5$ adders where each need only add together 5-bit numbers.

d. First note that the memory is byte addressed, thus when we store the 32-bit constant at address 0 we are actually storing the constituent bytes at addresses $0 \ldots 3$. Also note that the decimal value $305419896_{(10)}$ equals $12345678_{(16)}$.

A little-endian byte ordering convention places the least-significant byte at the lowest address and the most-significant byte at the highest address; thus one would expect to find the memory to look like

$$\begin{aligned} MEM[0] &= 78_{(16)} \\ MEM[1] &= 56_{(16)} \\ MEM[2] &= 34_{(16)} \\ MEM[3] &= 12_{(16)} \end{aligned}$$

after the instruction completes. A big-endian byte ordering convention uses the opposite ordering; it places the least-significant byte at the highest address and the most-significant byte at the lowest address to give

$$\begin{aligned} MEM[0] &= 12_{(16)} \\ MEM[1] &= 34_{(16)} \\ MEM[2] &= 56_{(16)} \\ MEM[3] &= 78_{(16)} \end{aligned}$$

e. If the number of registers is increased from 32 to 64 then one more bit per register operand will be required to address them; so each register operand will require 6 bits in an encoded instruction rather than 5. Assuming that the actual instructions are not changed, this will result in less opcode space in the instruction formats and hence the instruction set will be able to support less instructions. If the number of registers is decreased from 32 to 16 then one less bit per register operand will be required to address them and the converse is true: there will be more space in the instruction formats for the opcode. A larger (or richer) set of instruc-

tions (that support more addressing modes for example) means a given program can probably be shorter. Potentially this means the program can execute faster, if the micro-architecture can execute all instructions as fast as before, and that the code density is better. The converse is true of a smaller (or more sparse) set of instructions.

An additional implication for having less registers is that the programmer (or compiler) will be able to keep less data items in the fast registers and will need to keep more in the slower main memory; this is termed spilling in the context of compilers. So with less registers there will probably be more memory accesses which result in worse performance (and visa versa).

Questions from Chapter 6

27

a. MIPS stands for Millions of Instructions Per Second; it is calculated by measuring how many instructions are executed in a one second period. The advantages of this are that it is easy to calculate but is is not a good measure of performance since it ignores how much work gets done during execution: a processor executing many simple instructions per second might do exactly the same amount of work as another processor which executes fewer more complex instructions in the same time period. CPI stands for Cycles Per Instruction. The idea is to decompose the performance in to the product of how long each instruction type takes to execute and how often that instruction type is executed. Typically, a RISC processor will have a low CPI rating whereas a CISC processor will have a high CPI rating. The advantages are that we can now factor in how much work is being done by each instruction and ignore clock rate, the disadvantage is that it relies on an average program which clearly never exists in real-life.

b. We know that the overall CPI can be computed using:

$$\text{CPI} = \sum_{i=1}^{n} \text{FREQUENCY} \cdot \text{CYCLES}$$

so applying this formula to the initial numbers, we find:

ALU	frequency $\cdot$ cycles $= 0.5 \cdot 1 = 0.5$
Load	frequency $\cdot$ cycles $= 0.2 \cdot 5 = 1.0$
Store	frequency $\cdot$ cycles $= 0.1 \cdot 3 = 0.3$
Branch	frequency $\cdot$ cycles $= 0.2 \cdot 2 = 0.4$
Total	2.2

Using the same method on the numbers after the change, we find:

ALU	frequency $\cdot$ cycles $= 0.55 \cdot 1 = 0.55$
Load	frequency $\cdot$ cycles $= 0.10 \cdot 5 = 0.50$
Store	frequency $\cdot$ cycles $= 0.10 \cdot 3 = 0.30$
Branch	frequency $\cdot$ cycles $= 0.25 \cdot 2 = 0.50$
Total	1.85

The speed-up can be calculated as follows:

$$\text{SPEED-UP} = \frac{\text{NEW PERFORMANCE}}{\text{OLD PERFORMANCE}}$$

so we have that the new processor is 1.19 times faster than the old one, about a 16% improvement.

28

- Firstly, it is sometimes difficult to know what one should be measuring. Latency (i.e., execution time) is often the most important factor, but throughput for example is also a valid performance metrics. Without some idea of what the interesting feature of performance is, constructing a benchmark program to measure it is difficult.
- Next it is difficult, if not impossible, to find a suite of benchmark programs that perfectly represent "typical" workloads. Within a restricted domain (e.g., scientific or media kernels) this is more tractable, but generally one can always point at examples that are somehow atypical and hence which are not modelled by the benchmark.
- Even once the benchmark programs themselves are selected, the problem of input data is important. Processor features such as branch prediction and cache memories have data-dependant behaviour; this implies that benchmarking them with worst-case input will yield much different behaviour than best-case.
- Finally, it can often be difficult to actually perform measurements if a high degree of accuracy is required; often this requires support from the processor (e.g., performance counters) and long experimental trials to smooth over any local anomalies.

Following the description above, one might clearly "cheat" by using input data which is best-case in some way and hence does not represent typical use of the benchmark. Another approach might be to develop a special purpose compiler that behaves in some special way for the benchmark program. A compiler is supposed to be general purpose, but if it can make assumptions about the input (i.e., that the input is a particular benchmark program), it can potentially generate much more efficient output than one might expect.

Questions from Chapter 7

29

a. Clearly we can implement $\neq$ by negating the result of $=$ and likewise for $<$ and $\geq$, and $>$ and $\leq$. Furthermore, we can build $\geq$ from $>$ and $=$, and $\leq$ from $<$ and $=$. So essentially we only need two comparisons, say $=$ and $<$ to be able to compute the rest so long as we have the logic operations as well. The choice of which three is simply a matter of which ones you want to go faster: the ones built from a combination of other comparison and logic instructions will take longer to execute. One might take the approach of looking at C programs and selecting the set most used. For example $=$ and $<$ are used a lot to program typical loops; one might select them for this reason.

b. You can be as fancy as you want with any optimisations or special cases, for example checking for multiplication by zero, one or a power-of-two might be a good idea. But basically, the easiest way to do this is with the binary or double-and-add method:

```
short mul( short x, short y )
{
  short r = 0;
  short i = 0;

  switch( x )
  {
    case  0 : return 0;
    case  1 : return y;
    case  2 : return y << 1;
    case  4 : return y << 2;
    case  8 : return y << 3;
    case 16 : return y << 4;
  }

  switch( y )
  {
    case  0 : return 0;
    case  1 : return x;
    case  2 : return x << 1;
    case  4 : return x << 2;
    case  8 : return x << 3;
    case 16 : return x << 4;
  }

  for( i = 15; i >= 0; i-- )
  {
    r = r << 1;

    if( ( x >> i ) & 1 )
      r = r + y;
  }

  return r;
}
```

c. A basic implementation might look like the following:

```
short H( short x )
{
  short t = 0;
  short i = 0;

  for( i = 0; i < 16; i++ )
  {
    if( ( x >> i ) & 1 )
      t = t + 1;
  }

  return t;
}
```

but this has a number of drawbacks. Mainly, the overhead of of operating the loop and the content is quite high in comparison to the actual operation we are performing. The loop body for example is only a few instructions; it takes nearly as many to actually test and increment `i` during each iteration. Furthermore, the number of branches in the code means that pipelined processors might not execute this efficiently at all. An improvement is to use some form of divide-and-conquer approach where we split the problem into 2-bit then 4-bit chunks and so on. The result might look like:

```
short H( short x )
{
  x = ( x & 0x5555 ) + ( ( x >> 1 ) & 0x5555 );
  x = ( x & 0x3333 ) + ( ( x >> 2 ) & 0x3333 );
  x = ( x & 0x0F0F ) + ( ( x >> 4 ) & 0x0F0F );
  x = ( x & 0x00FF ) + ( ( x >> 8 ) & 0x00FF );

  return x;
}
```

30 First, note that the result via a naive method would be

$$\begin{aligned} r_2 &= x_1 \cdot y_1 \\ r_1 &= x_1 \cdot y_0 + x_0 \cdot y_1 \\ r_0 &= x_0 \cdot y_0. \end{aligned}$$

However, we can write down three intermediate values using three multiplications as

$$\begin{aligned} t_2 &= x_1 \cdot y_1 \\ t_1 &= (x_0 + x_1) \cdot (y_0 + y_1) \\ t_0 &= x_0 \cdot y_0. \end{aligned}$$

The original result can then be expressed in terms of these intermediate values as

$$\begin{aligned} r_2 = t_2 \qquad\qquad &= x_1 \cdot y_1 \\ r_1 = t_1 - t_0 - t_2 &= x_0 \cdot y_0 + x_0 \cdot y_1 + x_1 \cdot y_0 + x_1 \cdot y_1 - x_0 \cdot y_0 - x_1 \cdot y_1 \\ &= x_1 \cdot y_0 + x_0 \cdot y_1 \\ r_0 = t_0 \qquad\qquad &= x_0 \cdot y_0. \end{aligned}$$

So roughly speaking, over all we use three $(n/2)$-bit multiplications and four $(n/2)$-bit additions.

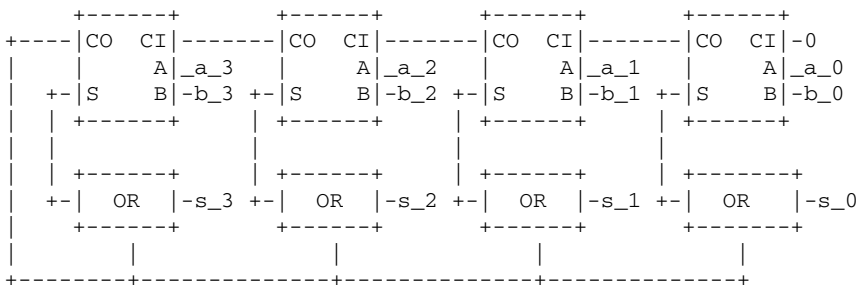
31

a. In binary, the addition we are looking at is

$$\begin{array}{rl} 10_{(10)} = & 1010_{(2)} \\ 12_{(10)} = & \underline{1100_{(2)}} + \\ & 10110_{(2)} \end{array}$$

where $10110_{(2)} = 22_{(10)}$. In 4 bits this value is $0110_{(2)} = 6_{(10)}$ however, which is wrong.

b. A design for the 4-bit clamped adder looks like



32 The relationship

$$x \text{ is a power-of-two} \leftrightarrow (x \wedge (x-1)) = 0$$

performs the test which can be written as the C expression

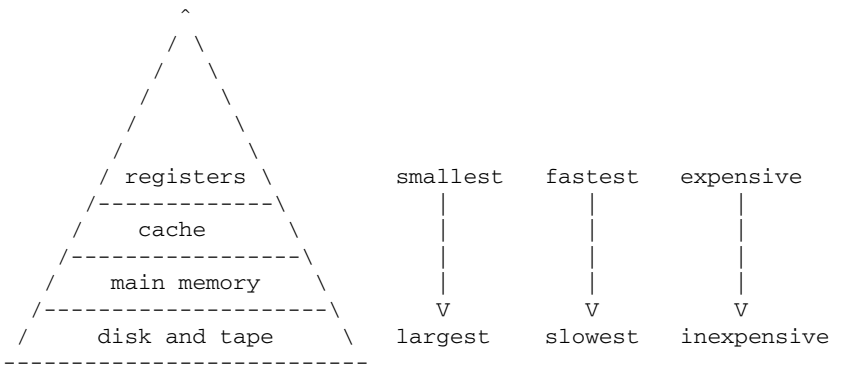
```
( x & ( x - 1 ) ) == 0
```

This works because if x is an exact power-of-two then $x-1$ sets all bits less-significant that the n-th to one; when this is ANDed with x (which only has the n-th bit set to one) the result is zero. If x is not an exact power-of-two then there will be bits in x other than the n-th set to one; in this case $x-1$ only sets bits less-significant than the least-significant and hence there are others left over which, when ANDed with x, result in a non-zero result. Note that the expression fails, i.e., it is non-zero, for $x = 0$ but this is okay since the question says that $x \neq 0$.

Questions from Chapter 8

33 The memory hierarchy is arranged like a pyramid with higher levels relying on the performance and functionality of lower levels. The idea is that higher levels are smaller and more expensive but faster, lower levels are large and inexpensive

but slower. The principles of locality means that we can keep the working set of a program, i.e., the data and instructions used most often, in the small, fast areas of the hierarchy. This means the most often used data can be accessed very quickly but we can still store very large amount of data since the hierarchy is backed by large, inexpensive storage.



At the top of the hierarchy are registers, these are word sized areas of on-chip memory used to produce operands for instructions. Typically they total around 1 kilobyte in size. The next level is cache memory, typically made from SRAM, which can be on or off-chip and ranges in size from around 512 kilobytes to several megabytes. Beyond any cache structures is the main RAM memory which is typically built from DRAM and is of the order of gigabytes in size. Finally, at the bottom of the hierarchy are the longer terms storage options such as disk and tape which are multi-gigabyte in size.

34

a. To address cache lines with 8 sub-words (i.e., 8 bytes in each line) we need a 3-bit word offset since $2^3 = 8$. There are 512 bytes in total which means there are $512/8 = 64$ lines; this means we need a 6-bit line number since $2^6 = 64$. This means there are $32 - 3 - 6 = 23$ bits remaining which form the tag.
b. To address cache lines with 8 sub-words (i.e., 8 bytes in each line) we need a 3-bit word offset since $2^3 = 8$. Since there are 64 lines, assuming that access in each of the $64/2 = 32$ sets is fully-associative, we need an 5-bit set number since $2^5 = 32$. This means there are $32 - 3 - 5 = 24$ bits remaining which form the tag.
c. To address cache lines with 8 sub-words (i.e., 8 bytes in each line) we need a 3-bit word offset since $2^3 = 8$. Since there are 64 lines, assuming that access in each of the $64/4 = 16$ sets is fully-associative, we need an 4-bit set number since $2^4 = 16$. This means there are $32 - 3 - 4 = 25$ bits remaining which form the tag.
d. To address cache lines with 8 sub-words (i.e., 8 bytes in each line) we need a 3-bit word offset since $2^3 = 8$. In a fully associative cache lines can be placed anywhere so there is no need to derive them from the address; this means there are $32 - 3 = 29$ bits remaining which form the tag.

35

- Compulsory misses are caused by the first access to some content held in memory. In a sense these seem unavoidable: before an element X is accessed by the program, how can the processor know it will be accessed and therefore prevent the cache miss ? The answer is to take a speculative approach and pre-fetch content into the cache that has a reasonable probability of use; if the speculation is correct then the associated compulsory miss is prevented, if it is incorrect then there will be a cache miss which is no worse than our starting point. The major disadvantage of incorrect speculation is the potential for cache pollution which can increase conflict misses. Thus, some balance needs to be arrived at between the two approaches.
- Capacity misses are misses that are independent of all factors aside from the size of the cache; essentially the only solution is to increase the cache size so a larger working set can be accommodated.
- Conflict misses are caused by the eviction of resident cache content by a newly accessed content that maps to the same cache line. The implication is that if element X is evicted by element Y, there has been a conflict in the address translation: since both map to the same line the access to Y causes a cache miss. This effect can continue in the sense that if X is accessed again later, it will cause another cache miss. The use of set-associative caches can reduce the level of contention in the cache by allowing conflicting addresses to coexist within a fully-associative set. Alternatively one might take a software approach and skew the addresses at which X and Y are placed so they no longer map to the same cache line.

36

a. There are 4 bytes per-line so we need 2 bits to address these since $2^2 = 4$. These are taken from the least-significant end of the address. There are 256 lines since the cache is 1 kilobyte in size; we need 8 bits to address these lines since $2^8 = 256$. These are again taken from the least-significant end, starting after the bits for the word address. The tag constitutes the rest of the bits so we can uniquely identify a given address in a line, it is therefore 22 bits in size.

b. Spacial locality is the tendency that if one item is accessed, those items close to it in memory are also accessed. For example if `A[1]` is accessed there is a good chance `A[0]` or `A[2]` will be accessed rather than say `A[1000]`. Temporal locality is the tendency that if one item is accessed, it will be accessed again in the near future. For example, when the add instruction which implements the addition of `B[i]` and `C[i]` is accessed, it has a high probability of being accessed again since it is in a loop. Cache interference is where items compete for space in the cache. If two items map to the same line and one is loaded into the line, an access to the other will evict or displace the original since the line cannot hold both.

c. The load access to `B[0]` will miss since this item is not in the cache. Such a miss means the item is loaded into cache line one, along with the values `B[1]`, `B[2]` and `B[3]`. The load access to `C[0]` also misses but since it maps to the

same line as `B[0]`, the data loaded evicts the values of `B` loaded in the previous load. Finally, the store to `A[0]` misses and again maps to the same line as `C[0]` so again evicts the current values. The loop continues in this manner, essentially never producing any cache hits because each access evicts the previous data.

d. i. This alteration changes matters entirely since now, `A[0]` maps to line 0, `B[0]` maps to line 1 and `C[0]` maps to line 2. As a result each access to `B[i]` will provoke one miss to load the values into the line, followed by three hits as the values `B[i+1]`, `B[i+2]` and `B[i+3]` will sit in the same line. The same is true for accesses to `C[i]` and `A[i]` so that the program will run much faster due to the decreased number of accesses to main memory.

ii. The basic problem here is one of cache interference: we want to prevent or reduce the probability of two addresses mapping to the same line in the cache and evicting each other. Associative caches are one solution to this problem; they allow any address to map to any line and include some decisional mechanism that places them. For example, least recently used (LRU) or random placement both solve the interference problem. The problem will associative caches is that they require significant resource to build. As a compromise, set-associative caches are a better option. A set-associative cache has number of sets each of which is like a small cache; inside each set look-up is fully associative but since the sets are small, the resulting cost is manageable. The idea is that we select the set a data item will go into using some more bits from the address: items that would map to the same line like `A`, `B` and `C` now map into the same line within different sets so don't interfere.

Questions from Chapter 9

37

a. The latency of a circuit is the time elapsed between when a single operation starts and when it finishes. The throughput of a circuit is the number of operations that can be started in each time period; that is, how long it takes between when two subsequent operations can be started.

b. The latency of the circuit is the sum of all the latencies of the parts:

$$40\text{ns} + 10\text{ns} + 30\text{ns} + 10\text{ns} + 50\text{ns} + 10\text{ns} + 10\text{ns} = 160\text{ns}.$$

The throughput is related to the length of the longest pipeline stage; since the circuit is not pipelined, this is $\frac{1}{160 \cdot 10^{-9}}$.

c. The new latency is still the sum of all the parts but now also includes the pipeline register:

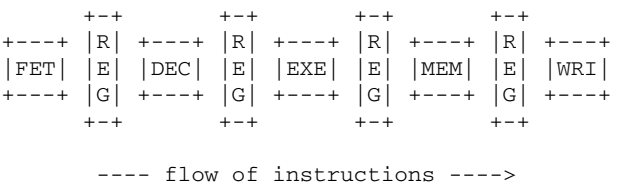
$$40\text{ns} + 10\text{ns} + 30\text{ns} + 10\text{ns} + 10\text{ns} + 50\text{ns} + 10\text{ns} + 10\text{ns} = 170\text{ns}.$$

However, the throughput is now more because the longest pipeline stage only has a latency of 100 ns including the register; the new throughput is thus $\frac{1}{100 \cdot 10^{-9}}$ which means we can start new operations more often than before.

d. To maximise the throughput we need to try and reduce the latency of all pipeline stages to a minimum. Since there is already a part with a latency of 50ns, the lower limit is 60ns if one includes the associated pipeline register. We can achieve this 60ns latency by placing two more pipeline registers between parts B and C and parts E and F. This has the effect of ensure all stages have at most a latency of 60ns and hence maximises the throughput but clearly has a disadvantage: the overall latency is increased to 190ns even though the throughput is $\frac{1}{60 \cdot 10^{-9}}$.

38

a. The idea is that the original data-path is split up into stages which match the sequential steps it takes. By operating these stages in parallel with each other, the pipeline can be executing instruction 1 while decoding instruction 2 and fetching instruction 3 for example. Although there are some caveats to this, for example dependencies between instructions might prevent the pipeline from advancing, generally this will improve performance significantly. For example, the instruction throughput of an n stage pipelined design will approach an n-fold improvement on the multi-cycle design given suitably written programs.
The basic design for a pipelined processor data-path can be split into five stages with pipeline registers between them to prevent a stage updating the working data for another stage:



The fetch stage will perform an instruction fetch, it will house the *PC* and an interface to the instruction memory and typically an adder to increment the *PC*. The decode phase decodes the instruction and fetches register operands, therefore it houses the *GPR*. The execution stage does the actual instruction execution and will therefore house the *ALU* that performs arithmetic and comparisons. The memory stage simply has to perform data memory access so only needs an interface to the data memory. Finally, the write-back stage houses no actual components but has hooks back into the *GPR* in the decode stage so it can write values. The *IR* is sort of spread along the whole pipeline in the pipeline registers; each stage is operating on a different instruction so each stage has its own *IR* in some sense.

b. A structural dependency is where two stages of the pipeline need to access a single hardware module at the same time. For example, this is commonly caused when one needs to increment the program counter and perform an ALU operation at the same time. A control dependency exists since the processor does not know

where to fetch instructions from until the branch is executed and so could, if not controlled, execute instructions from the wrong branch direction. A data dependency is where a computational instruction is being processed by the pipeline but the source values have not been generated or written-back to the register file.

c. The solution for structural dependencies is simple: one either replicates the hardware module so both competing stages have their own, dedicated device; or one ensures somehow that they use one single device in different phases of the same cycle. One way of dealing with control dependencies is to delegate the problem to the compiler and ensure that the delay slot following the branch is either filled with null operations whose execution does not effect either branch direction, or valid instructions scheduled from before the branch. We can also reduce the chance of stall by better predicting which direction the branch will take and hence lessening the chance of having to squash invalid instructions in the pipeline. A branch target buffer of some sort will usually accomplish this. Data dependencies are usually solved by forwarding results directly from within the pipeline back to the execution unit. Using this technique, the results don't need to be written-back before they are used during execution. Using these techniques, the processor seldom needs to stall as a result of any dependency although clearly they complicate the actual processor operation.

d. A branch delay slot is the space in time between when a branch is fetched and when it is executed. During this time the processor might have to stall since it cannot know instructions that would be fetched actually need to be executed. However, if the processor does not stall but just carries on fetching and executing instructions regardless, there is an opportunity to place instructions we want to execute in the branch delay slot. The advantage of doing so is that there is less wasted time but in order to realise this improvement we must be able to re-order instructions in the program so that the slot is filled; if we cannot fill the slot then NOPs must be placed there instead to ensure correct execution. This all means that the processor can be simpler due to the lack of stall condition, but the programmer must do more work to extract the improvement in performance and ensure correctness.

   In most RISC architectures, alteration of control-flow is done explicitly using dedicated branch instructions. Predicated execution offers an alternative to this model by associating a predicate function to each instruction and only executing the instruction if the predicate is true. Rather than set the *PC* to some new value, this approach allows one to skip over instructions based on the associated predicate. It is ideal for conditional execution of short sections of code due to the low overhead associated. In this case, both the processor and programmer must be more complicated; the processor must be able to decode and evaluate predicates and discard instructions which fail, the programmer must be able to write programs in a style that takes advantage of this feature.

   Branch prediction reduces the number of stalls by guessing which way a branch will complete and continuing to fetch instructions from that path; if the prediction is wrong then the fetched instructions must be squashed but if it is right execution can continue with no loss of time. Common prediction mechanisms are simple,

for example one might predict that backwards branches (that model loops) are always true: this will be correct most of the time because a loop only exists once and iterates many times. The complexity in branch prediction is entirely in the processor; the programmer need do nothing more than normal unless the ISA allows hints to be passed about the probably direction of branches.

39 After "partially" unrolling the loop by a factor of four, it is clear that the four additions that represent the new loop body are independent. With a vector processor, either dedicated or in the form of an extension to a conventional processor (e.g., MMX/SSE), these can be executed in parallel. Starting with the scalar implementation

$$
\begin{array}{ll}
 & \ldots \\
 & GPR[1] \leftarrow 0 \\
 & GPR[2] \leftarrow 400 \\
loop: & \text{if } GPR[1] \geq GPR[2], PC \leftarrow exit \\
 & GPR[3] \leftarrow MEM[\mathtt{B} + GPR[1]] \\
 & GPR[4] \leftarrow MEM[\mathtt{C} + GPR[1]] \\
 & GPR[5] \leftarrow GPR[3] + GPR[4] \\
 & MEM[\mathtt{A} + GPR[1]] \leftarrow GPR[5] \\
 & GPR[1] \leftarrow GPR[1] + 1 \\
 & PC \leftarrow loop \\
exit: & \ldots
\end{array}
$$

the basic idea would be to utilise the vector register file and associated ALUs to load and execute four additions at once. This could be described as

$$
\begin{array}{ll}
 & \ldots \\
 & GPR[1] \leftarrow 0 \\
 & GPR[2] \leftarrow 400 \\
loop: & \text{if } GPR[1] \geq GPR[2], PC \leftarrow exit \\
 & VR[1] \leftarrow MEM[\mathtt{B} + 0 + GPR[1] \ldots \mathtt{B} + 3 + GPR[1]] \\
 & VR[2] \leftarrow MEM[\mathtt{C} + 0 + GPR[1] \ldots \mathtt{C} + 3 + GPR[1]] \\
 & VR[3] \leftarrow VR[1] + VR[2] \\
 & MEM[\mathtt{A} + 0 + GPR[1] \ldots \mathtt{A} + 3 + GPR[1]] \leftarrow VR[3] \\
 & GPR[1] \leftarrow GPR[1] + 4 \\
 & PC \leftarrow loop \\
exit: & \ldots
\end{array}
$$

whereby we get roughly a 4-fold improvement in performance as a result. Of course this depends on being able to load and store operands quickly enough (this is assisted by the regular nature of such accesses), and that there are enough ALUs to execute the parallel additions.

The case for VLIW is more complicated, in part because such an architecture is less well defined. Imagine we have a VLIW processor which can accept instruction bundles that include two memory accesses and one arithmetic operations. The idea here would be to software pipeline the loop so that we "skew" the iterations: this allows us to load the operands for iteration 1 at the same time as performing the

addition for iteration 0 for example. Reading operations on the same row as being executed in parallel (the first two columns being memory access and the third being arithmetic), an example code sequence is show in Figure B.1. Notice that by overlapping the loop iterations, we are roughly executing one iteration in two instruction bundles rather than via five conventional instructions as above (even though there are some empty slots in selected bundles).

40

a. Consider an example function which sums elements from two arrays and stores the result into a third:

```
void add( int* A, int* B, int* C, int p )
{
  for( int i = 0; i < p; i++ )
    A[ i ] = B[ i ] + C[ i ]
}
```

On a scalar processor, each iteration of the loop is executed separately. The use of processor designs such as superscalar might capitalise on the implicit parallelism between loop iterations but essentially any given operation only deals with one (scalar) data item at a time. The idea of a vector processor is to make this parallelism explicit and equip the processor with the ability to operate on many data items (vector) using a single operation. As such, instead of executing many scalar operations of the form

$$\begin{aligned} A[0] &\leftarrow B[0]+C[0] \\ A[1] &\leftarrow B[1]+C[1] \\ A[2] &\leftarrow B[2]+C[2] \\ &\vdots \\ A[p-1] &\leftarrow B[p-1]+C[p-1] \end{aligned}$$

the vector processor executes one vector operation of the form

$$A[0\ldots p-1] \leftarrow B[0\ldots p-1]+C[0\ldots p-1].$$

To achieve this, the processor needs two key components: a vector register file capable of storing vectors (i.e., many data items rather than single data items), and a number of processing elements each of which operates on one element of the input vectors to produce one element of the output vector. The processing elements can be sub-pipelined if if there are less of them than there are elements in a vector, they may need to be used iteratively to operate across the entire vector content.

Typically these components are gathered together into a single co-processor type unit which is invoked by a standard scalar processor (i.e., rather than use the scalar register and ALU) when vector instructions are encountered. This approach needs careful management of memory access so that a single interface to memory can be used effectively to load and store vector data.

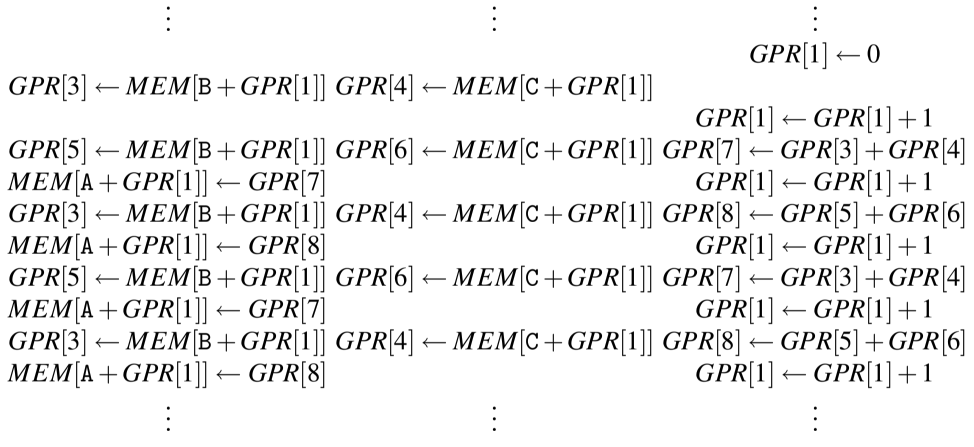


Figure B.1 A VLIW implementation of the vector addition loop body.

r_0	(	D	,	C	,	B	,	A	)			
r_1	(	$2D$	,	$2C$	,	$2B$	,	$2A$	)	$add(r_0, r_0)$		
r_2	(	$2D+1$	,	$2C+1$	,	$2B+1$	,	$2A+1$	)	$add(r_1, 1)$		
r_3	(	$D(2D+1)$	,	$C(2C+1)$	,	$B(2B+1)$	,	$A(2A+1)$	)	$mul(r_2, r_0)$		
r_4	(	$rotl(D(2D+1),5)$	,	$rotl(C(2C+1),5)$	,	$rotl(B(2B+1),5)$	,	$rotl(A(2A+1),5)$	)	$rotl(r_3, 5)$		
		At this point we rename the sub-words to make life easier										
r_4	(	D'	,	C'	,	B'	,	A'	)			
r_5	(	C'	,	D'	,	A'	,	B'	)	$shuffle(r_4)$		
r_6	(	C'	,	B'	,	A'	,	D'	)	$shuffle(r_4)$		
r_7	(	$D \oplus C'$	,	$C \oplus D'$	,	$B \oplus A'$	,	$A \oplus B'$	)	$xor(r_5, r_0)$		
r_8	(	$rotl(D \oplus C', C')$	,	$rotl(C \oplus D', B')$	,	$rotl(B \oplus A', A')$	,	$rotl(A \oplus B', D')$	)	$rotl(r_7, r_6)$		
r_9	(	0	,	$K[i+1]$	,	0	,	$K[i+0]$	)	$load$		
r_{10}	(	$rotl(D \oplus C', C')$	,	$rotl(C \oplus D', B') + K[i+1]$	,	$rotl(B \oplus A', A')$	,	$rotl(A \oplus B', D') + K[i+0]$	)	$add(r_8, r_9)$		
r_{11}	(	A	,	C	,	D	,	B	)	$shuffle(r_0)$		
r_{12}	(	$rotl(B \oplus A', A')$	,	$rotl(D \oplus C', C') + K[i+1]$	,	$rotl(A \oplus B', D')$	,	$rotl(C \oplus D', B') + K[i+0]$	)	$shuffle(r_8)$		
r_{13}	(	$rotl(A \oplus B', D')$	,	D	,	$rotl(C \oplus D', B')$	,	B	)	$unpack(r_{11}, r_{12})$		
		At this point we rename the sub-words to make life easier										
r_{11}	(	A''	,	D''	,	C''	,	B''	)			

Figure B.2 A vector implementation of the RC6 loop body.

b. The use of a vector processor can yield higher performance than a scalar processor (for a selected class of programs) because:

   - Parallelism in the program is explicit. The processor can therefore take advantage of this parallelism, with very little effort or overhead in terms of hardware complexity, by using multiple processing elements. That is, processing element i can operate on element i of the input vectors to produce element i of the output at the same time as other elements $j \neq i$ are being produced in parallel. Consider a processor that operates on vectors of length p with r processing elements; the processor achieves a theoretical peak performance improvement of p/r over the scalar processor if the program is easily vectorised.
   - In scalar, multiple issue architectures such as superscalar, the demand for new instructions is very high: they must be fetched from memory fast enough to satisfy the aim of issuing one operation per-cycle. Although cache memories and other techniques can help, this can present a problem. In contrast, the amount of work done by per-operation in a vector is much higher (e.g., a factor of p higher if we work with vectors of length p). As such, the pressure for operations to be fetched quickly is less and hence the impact of the bottleneck represented by main memory is less pronounced.
   - Access to vector data held in memory is typically more regular than that to scalar data. That is, if we load a p element vector at address A, we explicitly load from addresses $A+0 \ldots A+p-1$. The memory hierarchy can capitalised on this feature via specialist cache design and operational policies, prefetch and striding, burst access and so on. In the case of access to scalar is not as easy to deal with: single accesses are disjoint and cannot be easily related to the overall task of access to the larger vector.

c. Say we represent the input at the start of the loop body as (D,C,B,A), i.e., a packed 128-bit register of 32-bit sub-words. We assume that the array K has been padded with space so that when elements are loaded during iteration i, we get the vector $(0,K[i+1],0,K[i+0])$. Finally, we assume that all computation is performed component-wise on sub-words (i.e., using strict SIMD) but that there are a number of standard sub-word organisation operations such as shuffle and pack. Equipped as such, we can implement the loop body using the operation sequence in Figure B.2 so that at the end, we have the vector (A'',D'',C'',B'') which is used as input to the next iteration.

d. Implementation of the loop body using 3-address style RISC operations can be achieved using the following sequence

$$
\begin{aligned}
t_0 &\leftarrow B + B \\
t_1 &\leftarrow t_0 + 1 \\
t_2 &\leftarrow B \cdot t_1 \\
t_3 &\leftarrow rotl(t_2, 5) \\
t_4 &\leftarrow D + D \\
t_5 &\leftarrow t_4 + 1 \\
t_6 &\leftarrow D \cdot t_5 \\
t_7 &\leftarrow rotl(t_6, 5) \\
t_8 &\leftarrow A \oplus t_3 \\
t_9 &\leftarrow rotl(t_8, t_7) \\
t_{10} &\leftarrow MEM[(K+0)+i] \\
t_{11} &\leftarrow t_9 + t_{10} \\
t_{12} &\leftarrow C \oplus t_7 \\
t_{13} &\leftarrow rotl(t_{12}, t_3) \\
t_{14} &\leftarrow MEM[(K+1)+i] \\
t_{15} &\leftarrow t_{13} + t_{14} \\
A &\leftarrow B \\
B &\leftarrow t_{13} \\
C &\leftarrow D \\
D &\leftarrow t_8
\end{aligned}
$$

This scalar sequence requires 20 operations; the vector equivalent requires 13 operations. However, the four move operations at the end of the scalar implementation are essentially just renaming intermediate variables: the actual moves are not required. Therefore by unrolling the loop somewhat it should be possible to remove these and reduce the operation count to 16. Although the sequences are not scheduled intelligently, it seems possible that with some rearrangement (plus possibly some software pipelining) it is possible to execute roughly one operation per-cycle. As such, the vectorisation has produced roughly an 19% improvement in performance (versus roughly 35% without unrolling).

The theoretical limit is a four-fold improvement in performance since one would expect roughly four operations to occur in parallel in the vector case. The two main factors which prevent the improvement being higher are wasted work in computing values we don't need (e.g., the $rotl(D(2D+1),5)$ sub-word of r_4), and the amount of shuffle type operations required to move sub-words into the right place for actual computation. In short, the loop looks easily vectorisable but the architecture (i.e., packing four 32-bit sub-words into 128-bit registers) does not capitalise on this well. One might change the processor to pack two 32-bit sub-words into 64-bit registers; this would prevent wasted computation and reduce either the amount of processing elements required if there is one processing element for each sub-word (e.g., two rather than four), or the time taken for each operation if there are less processing elements than sub-words (e.g., in the initial design, having two processing elements rather than four means iterating twice for each operation).

41

a. Speculative execution is a mechanism that acts to reduce the cost of branches. The problem caused by branches is that until any branch condition and target are known, the processor cannot proceed: it cannot know where to fetch the next instructions from and hence the front-end becomes stalled. In turn, this can lead to under utilisation of the computational resources which are idle until provided with work.
The basic idea is to construct a group of components (that basically guess which way a branch will direct control-flow and execute instructions from that path. If the guess turns out to be correct, then execution can continue more or less as normal: the instructions allowed into the pipeline really were those the programmer intended to execute so in a sense no execution time has been wasted. If the guess turns out to be incorrect however, the instructions in the pipeline are invalid (i.e., they are from the wrong path) and need to be deleted. After cleaning up, execution is restarted along the correct path and although there is a associated cost, this is not significantly more than the case without speculation. As a result, as long as guesses made by the processor are correct more often than not, performance is improved because more often than not the front-end does not stall and can supply a more consistent stream of work to the back-end.
Note that use of a re-order buffer makes speculation much easier: we can simply include a flag in the RoB to mark speculatively executed instructions and only allow them to complete (i.e., exit the RoB) if the associated condition turns out to be true.

b. The case of static (rather than fixed) branch prediction demands that the compiler and/or programmer does some analysis of the program to work out the probable control-flow. For example, the compiler might:

   - Examine a particular loop statement and reason that since the loop bound is large, a branch from the end of the loop body back to the start is much more likely to be taken than not taken.
   - Examine a particular conditional statement and reason that the condition is very unlikely to evaluate to false, and hence that a branch that skips the conditional body is very unlikely to be taken.

   The idea is to pass the results of this analysis to the processor in order to guide any decisions. One mechanism to do this is to allow "hints" or predicates in the branch instructions that inform the processor that they are likely or unlikely to be taken: based on such a "hint", the processor might speculatively execute one control-flow path rather than another. Disadvantages of this approach is include the fact that data-dependent behaviour of the program is not considered (unless we use some form of profiling), and that there is a limited amount of space in any instruction encoding with which to accommodate the "hints".
   Use of dynamic branch prediction means that the processor tries to perform analysis of the program behaviour at run-time. Although this requires extra hardware to implement the scheme, it means that any data-dependent behaviour can be detected and capitalised upon. Typically the processor would include a mechanism for predicting whether a branch is taken or not *and* where the branch target is. The

branch target is possible simpler: a cache-like structure called the Branch Target Buffer (BTB) can be maintained whereby the address of a branch is mapped to a predicted target. If the branch address has no mapping in the BTB then the processor simply has to guess, but if there is a mapping (e.g., the branch was executed recently) then the processor can speculatively use the previous target. Predicting the direction of a branch can be achieved using a Branch Prediction Buffer (BPB) which maps the address of a branch to a history of that branches behaviour, i.e., a history of whether it was taken or not. Based on this history, one can make predictions about whether it will be taken or not in the future. Examples include the use of saturating counters which effectively treat the problem like a state machine where the branch history represents the current state (which acts as the prediction), each executed branch updates the state in a manner that depends on the real behaviour.

c. i. Assuming for example that the "call" and "return" operate in a similar way to those in MIPS, a simple calling convention could be:

```
caller: push arguments
        call callee
        pop  return value
        pop  arguments
        ...

callee: push return address
        push space for locals
        ...
        pop  space for locals
        pop  return address
        push return value
        return to caller
```

In this case, the "call" places the return address in a specific register (this is a so-called "jump and link") while the "return" is implemented via a branch to an address held within any register: the address is popped from the stack into a register and then used.

ii. The "call" instruction is essentially an unconditional branch: it does not require any branch prediction since it is *always* taken and the branch target is *always* known if it can be encoded as an immediate operand. The "return" instruction is more tricky: although unconditional, in this case the processor will not know the branch target (i.e., the return address) until it is retrieved from the stack.

One approach would be to employ a system whereby there is segregation between data and address related stack content, i.e., use a dedicated return address stack. This would require the "call" instruction to push the return address onto the address stack and the "return" instruction to pop it: these operations could form part of the instruction semantics. Since the address stack will probably be more constrained in terms of size and use, it could be possible to retain

some of it on-chip (rather than in memory) so that access to the last say n return addresses is very fast.
Although there are potentially some disadvantages, the main advantage is that we can now "snoop" the address stack for a return address. That is, as soon as the "return" instruction is fetched we can look at the on-chip address stack and retrieve the associated branch target more quickly.

Questions from Chapter 10

42

a. This description represents a form of delay slot: the load operation takes some time to complete, say n cycles. Thus, the value loaded only becomes valid n cycles after the load seems to have executed and can therefore only be read after then. The alternative would be to either elongate the clock period so a load could complete in one cycle (this penalises instructions which are faster), or to stall the processor while the load is completed.
b. By placing the burden on the programmer or compiler, the processor design can be simpler: if it is pipelined there could be no need for some forms of forwarding logic and control is generally easier. This could lead to improvements in size and power consumption. In a sense the situation is more flexible in the sense that the programmer or compiler can make use of the delay slot to do other things (i.e., computation not related to the load). However, the programmer or compiler is faced with a harder task: if they make a mistake the program will execute incorrectly, plus porting the program to a processor with a different delay is made harder.

43

a. The arguments `$a0` and `$a1` represent the addresses of two arrays, say A and B, where elements in the array are 8-bit bytes:

- Instructions #1 to #4 load elements of A and inspects them: if the loaded element is equal to 0 then the function branches to `l1`, otherwise it increments `$a0` and branches back to `l0`. The end result is that once control-flow reaches `l1`, `$a0` holds the address of the first element in A which is zero.
- Instruction #5 stores the value of `$a1`, i.e., the address of B, in `$v0`.
- Instructions #6 and #7 load an element from B and store it at the end of A; if the element is zero, instruction #8 branches to `l3`. Otherwise, instructions #9 and #10 increment `$a0` and `$a1`, i.e., the addresses of A and B, then instruction #11 branches back to `l2`. The end result is that once control-flow reaches `l3`, the content of array B has been appended to the end of A (including the first zero element).
- Instructions #12 and #13 compute the return value into `$v0` by subtracting the previously stored value in `$v0` from the address of the last element stored at

the end of *A* (correcting the off-by-one error); this represents the total number of elements of *B* appended to the end of *A*.
- Finally, instruction #14 directs control-flow back to the caller using the return address in `$ra`.

Effectively, this is an implementation of the `strcat` function which concatenates two zero terminated strings (i.e., appends one string to the end of another).

b. The obvious optimisation is to use `lw` and `sw` to append four bytes of *B* to the end of *A* in each iteration of instructions #6 and #7; since the memory hierarchy can potentially service such access at the same speed as access to bytes, this potentially gives around a 4-fold improvement in performance (the loop is executed four times less). However, the problem is that if *B* is not a multiple of four characters long the terminating zero character will occur within the 32-bit word loaded by `lw`; this makes it harder to check when the terminator is encountered, and to avoid writing too much content to the end of *A* (and potentially overwriting other, legitimate data).

Questions from Chapter 11

44 The rough purpose of stack frame is to capture the run-time "state" of a function; it provides an area of memory where values relating to said state can be stored, and a mechanism to create and destroy sub-frames so that nested function calls are possible. The stack starts at some fixed address and grows downward; it is managed as a FIFO data-structure. This management is achieved using two pointers which can be held in general purpose registers and accessed using standard instructions, or held in special purpose registers and accessed using special instructions:

- The Stack Pointer (*SP*) keeps track of the Top of Stack (ToS); either it is the address of the next free space on the stack, or the first used space. The value of *SP* changes during execution of functions as items are pushed onto or popped from the stack.
- The Frame Pointer (*FP*) keeps track of the fixed base address of the current stack frame; this allows access to the stack frame content using a fixed offset (rather than a variable offset based on the moving address held in *SP*).

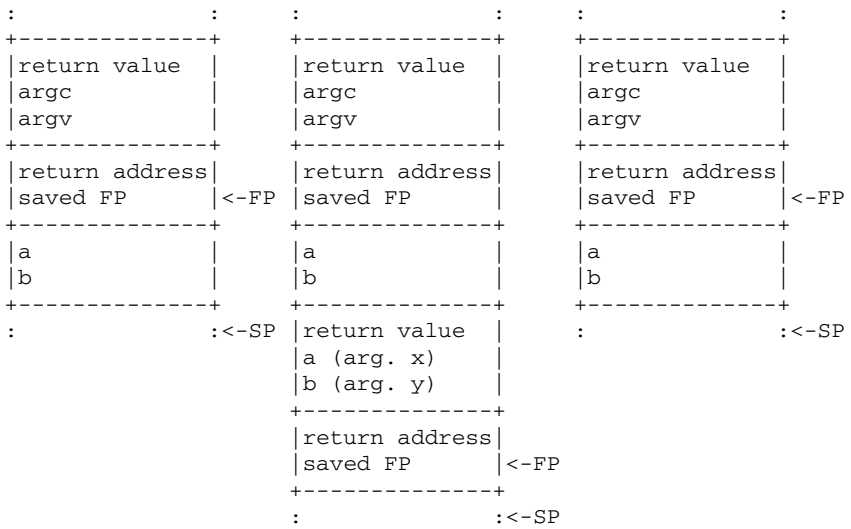
A given frame will, depending on the function, contain many features:

- Arguments passed from a caller function to a callee function can be pushed onto the stack by the caller and then retrieved by the callee. The cost of this can be reduced by passing arguments in registers; typically we include the return value as an "argument" or sorts.
- The return address is where a callee function should branch back to in order to continue execution from the point it was called from in the caller function.
- The saved frame pointer preserves the previous frame pointer value, i.e., the frame pointer of the caller, when the callee initialises a new stack frame and

hence a new frame pointer. This allows valid execution when control-flow returns to the caller.
- Within a function, if there are not enough registers to hold the working set of values some might need to be spilt into memory; the stack frame represents a good place for spilling to occur into. As such, part of the stack is reserved for "temporary values".
- Since the caller should be able to continue execution once the callee function has completed, the state of the caller should be maintained across calls to many callee functions. The task of saving this state is typically shared between the caller and callee: so-called caller save registers should be saved by the caller if their useful content might be overwritten, callee save registers are the responsibility of the callee function. In both cases, the register content is stored on the stack and the retrieved later when execution of the function is complete.

45

a. Assuming a calling convention which includes a frame pointer, and that there is no need for caller or callee saving of registers, the three cases could roughly be described by:



The "before" and "after" cases are clearly the same by design; the only side-effect of calling the function is the generation of a result (rather than alteration of the state somehow).

b. An implementation of the `max` function, and the `main` function that calls it, is as follows:

```
main: addi    $t0, $0,   10 # set a = 10
      addi    $t1, $0,   20 # set b = 20
```

```
        addi    $sp, $sp,  -4 # push rv
        sw      $t0, 0($sp)   # push a
        addi    $sp, $sp,  -4
        sw      $t1, 0($sp)   # push b
        addi    $sp, $sp,  -4

        jal     max           # call max(a,b)

        addi    $sp, $sp,   4 # pop b
        addi    $sp, $sp,   4 # pop a
        addi    $sp, $sp,   4 # pop rv
        lw      $t0, 0($sp)

        ...

max:    addi    $sp, $sp,  -4 # make ra space
        sw      $ra,  0($sp)  # push ra
        addi    $sp, $sp,  -4 # make fp space
        sw      $fp,  0($sp)  # push fp
        add     $fp, $sp, $0  # set  fp to sp

        lw      $t0, 16($fp)  # load x
        lw      $t1, 12($fp)  # load y

        sub     $t2, $t0, $t1 # compare
        bgtz    $t2, l0
        sw      $t1, 20($fp)  # set  rv = y
        b            l1
l0:     sw      $t0, 20($fp)  # set  rv = x

l1:     lw      $fp,  0($sp)  # pop  fp
        addi    $sp, $sp,   4 # free fp space
        lw      $ra,  0($sp)  # pop  ra
        addi    $sp, $sp,   4 # free ra space
        jr      $ra           # return
```

46 This is quite an open question, so emphasis is placed on level of background knowledge and quality of reasoning; probably some degree of background research is required. Taking ARM addressing modes as a concrete processor design, one might point to the addressing modes as supporting the types of operation required in typical C programs:

- The base/offset addressing mode is ideal for access to structures since one can use a pointer to the structure as the base, and then add the fixed distance of a field in the structure as the offset. This allows the access to be performed without the overhead of manual pointer arithmetic.
- The ability to scale the second operand to all instructions means one can scale the offset in a base/offset style access. This is ideal for access to arrays since it allows one to accommodate arrays of different C types without the overhead of manual scaling. That is, for an array of 8-bit elements no scaling is required; for an array of 16-bit elements one can scale the offset by two and so on.

- The ability to perform auto-indexing supports the update of loop counters required for most loop constructs. That is, the addressing mode updates the loop counter without the overhead of an extra addition.

Example topics for discussion wrt. other languages might include:

occam A lineage of Transputer processor designs have offered support for:

a. Light-weight, explicit parallel programs via low-cost fork, join and synchronisation operations for threads.
b. Efficient inter-thread communication via concrete, hardware based "channels" (effusively similar to managed buffers), that are mapped onto by abstract channels in a program.

Java a. Certain variants of ARM processor include support for "native" bytecode execution; for example they fetch and decode bytecode instructions and translate them into sequences of ARM instructions (including use of "special" registers).
b. Some specialist architectures have included support for types of "dynamic dispatch" that result from the use of virtual functions. The basic idea is to reduce the cost of the several memory accesses, which would be required in a naive case, that form the function address for a given type.

Pascal a. P-code and similar dedicated virtual instruction sets akin to Java bytecode, designed to execute compiled Pascal programs and sometimes realised in physical hardware.

Lisp a. Lisp machines, general purpose processors with dedicated hardware support for execution of Lisp programs.
b. Initiated by MIT Lisp Machine Project, such processors accelerated costly aspects of execution such as garbage collection and type checking (the later supported via "fat" values which included extra bits to represent type).

Forth a. Dedicated stack processors with data and control stacks and an instruction set to access them.
b. Optimisation examples include support to hold the top of stack value(s) on-chip rather than in memory.

47 The basic problem is that using instructions available in MIPS32, we would have to perform at least two comparisons and a branch in addition to each memory access. This will have an impact on performance and as such, is an ideal candidate for mitigating alterations to the processor.

One approach would be to include the concept of predicated execution and somehow capture the bounds check as a predicate that one can use to skip over the access. Another approach would be to add an addressing mode that is similar to a base/offset type scheme but includes a third operand that specifies some limit. Again in relation to ARM, we might take the `str` instruction that stores a single register at the effective address derived from a base and an offset. Using this, we might include an alternative called `strbc` (for "store register with bounds check") whose semantics are

$$\texttt{strbc}\ a,b,c,d = \begin{cases} MEM[b+c] \leftarrow GPR[a] & \text{if } 0 \leq b+c < d \\ \bot & \text{otherwise} \end{cases}$$

Using this instruction we can capture the conditional statement and the associated store with one instruction; this eliminates the cost of fetching, decoding and executing the conditional and improves the situation further by eliminating the associated branch. Note that there is however some burden placed on the programmer or compiler: they must be able to use the instruction effectively; this demands for example that the size of arrays (i.e., the bound d) is known whenever an array access is performed. This could be problematic in C where there is some duality between an array and a (more arbitrary) pointer.

Questions from Chapter 12

48 During a context switch, the state of one process is saved to the related Process Control Block (PCB) within the kernel address space, and another process state is restored; when kernel is finished, the second process is able to continue execution whereas before it started the first process was executing. A dominant cost within this operation is the transfer of state to and from memory: among other things, this cost is influenced by storing every register and then loading them again. One way to accelerate this is to permit "block" load and store instructions. An example of this approach is provided by ARM processors: the `ldm` and `stm` instructions accept a list of registers as operands. This list is either loaded from or stored to memory starting at a base address specified by another operand. The advantage of this is 2-fold: firstly one can take advantage of any burst accesses mode the memory provides, secondly the overhead relating to repeated fetch and decode of load and stores is reduced to a single fetch and decode for one `ldm` or `stm` instruction.

49

a. The resident set is that portion of a processes address space that is held in physical memory at a given point in time; the working set is the portion that is actually needed for execution. Typically the working set is an instantaneous measure in the sense that at different points in time, the working set will change due to the behaviour of the program. Clearly the two measures may differ: physical memory may not be able to accommodate the full working set, or may hold resident content which is not required.
b. The term "thrashing" relates to a situation where many page faults occur; due to contention (or simply lack of space) pages are being transferred into and out of physical memory often enough that this activity dominates useful computation. To detect this situation, the OS might monitor the page fault rate (e.g., the average number of page faults per-access to memory); when this reaches some threshold, it can state that "thrashing" occurs. Essentially the situation is a product of not being able to capture the working set in physical memory. If there is one process executing, this is somewhat unavoidable but if there are many processes

one might attempt to reduce the level of multitasking (i.e., prevent some processes becoming active). The hope is that by reducing the number of processes that compete for space in physical memory, the working set of those that remain will fit. So basically the idea would be to terminate or suspend low-priority or resource hungry processes until the "thrashing" situation is resolved.

c. Some advantages are:

- A reduction in the page table size; if there are less pages there are less potential entries in the page table.
- An increase in performance of the TLB; if there are less pages, there is more chance that the working set can be captured in the TLB and hence that there will be a TLB-hit.

Some disadvantages are:

- An increase in the time taken to transfer pages in and out of physical memory from the swap file; if the page is larger, it will naturally take longer to load it from disk.
- A typical process will not be allocated an amount of memory that is an exact multiple of the page size; part of the "last" page allocated to it will be unused. If the pages are larger, this unused region (which is wasted) is larger.

50

a.
- The user may have requested that the process be terminated.
- The process could have causes some unresolvable exception (e.g., out of bounds memory access, execution of an invalid instruction).
- The process could exit normally; although this is perhaps not an "abnormal" termination, from the perspective of the operating system the process is still terminated in some sense.
- The parent process (i.e., the process that forked the one we are interested in) may have terminated.

b.
- The user may have requested that the process be suspended.
- The process could have been suspended as a result of time-slicing; once the time slice of an executing process expires, it needs to be suspended.
- The process could have been suspended in an attempt to resolve "thrashing".
- The process could have been suspended pending some high-latency I/O operation (e.g., network communication or virtual memory management).

Questions from Chapter 13

51

a. A basic approach would be to use c as an index into an array which we initialise with the right value. For example, the C function would be

```
uint16 choose( uint16 c, uint16 y, uint16 z )
{
  uint16 T[2];
  T[0] = z;
  T[1] = y;
  return T[c];
}
```

Depending on the characteristics of memory access, this might be slow however: although it is straight-line and hence does not suffer from pipeline stalls from branch instructions, it includes three memory accesses (depending on the addressing modes in the processor. An improvement might be constructed by first considering a mask value $m = c - 1$ so that if $c = 0$ then $m - 1$ and if $c = 1$ then $m = 0$. Because of the way the twos-complement representation works, if $c = 0$ then m is a value where all the bits are set to 1 and if $c = 0$ then m is a value where all the bits are set to 0. We can use this mask value to select the right result from y and z using the logical expression

$$x = (m \wedge z) \vee (\neg m \wedge y).$$

That is, if $c = 0$ then m is a value with all bits set to 1 and we have that $m \wedge z = z$ and $\neg m \wedge y = 0$ so $(m \wedge z) \vee (\neg m \wedge y) = z$. Conversely, if $c = 1$ then m is a value with all bits set to 0 and we have that $m \wedge z = 0$ and $\neg m \wedge y = y$ so $(m \wedge z) \vee (\neg m \wedge y) = y$. So we can implement the function in C as

```
uint16 choose( uint16 c, uint16 y, uint16 z )
{
  uint16 m = c - 1;
  return ( m & z ) | ( ~m & y );
}
```

This is both straight-line and memory access free; it can be executing using only five instructions.

b. A basic approach might be to find an addition chain for 15 so that we can decompose the multiplication into a series of additions. By forcing summands to be powers-of-two (which can be efficiently generated using shifts), we can implement the operation as

```
uint16 mul( uint16 x )
{
  return ( x << 3 ) +
         ( x << 2 ) +
         ( x << 1 ) +
         ( x      ) ;
}
```

such that the return value is $x \cdot 2^3 + x \cdot 2^2 + x \cdot 2^1 + x \cdot 2^0 = 8x + 4x + 2x + 1x = 15x$. However, a more efficient addition chain can be constructed by noticing $15 = 16 - 1$; since we can generate $16x$ efficiently, we can re-write our solution as

```
uint16 mul( uint16 x )
{
  return ( x << 4 ) -
         ( x      ) ;
}
```

which uses two less shifts and two less additions.

c. The basic approach here is to iterate through all the bits in x testing each one and incrementing a counter if $x_i = 1$. For example, the C function would be

```
uint16 H( uint16 x )
{
  uint16 t = 0;

  for( int i = 0; i < 16; i++ ) {
    if( ( x >> i ) & 1 ) {
      t = t + 1;
    }
  }

  return t;
}
```

However, this is inefficient for a number of reasons. Specifically, the cost of operating the loop is high in comparison with the cost of the loop body: the overhead is significant. Furthermore, the instruction sequence required will include several branches which might stall the processor pipeline and degrade performance. A better approach is the straight-line C function

```
uint16 H( uint16 x )
{
  x = ( x & 0x5555 ) + ( ( x >> 1 ) & 0x5555 );
  x = ( x & 0x3333 ) + ( ( x >> 2 ) & 0x3333 );
  x = ( x & 0x0F0F ) + ( ( x >> 4 ) & 0x0F0F );
  x = ( x & 0x00FF ) + ( ( x >> 8 ) & 0x00FF );

  return x;
}
```

This works using a divide-and-conquer approach; each line of the function is counting bits within a sub-set of the original value. We mask x with the constant $5555_{(16)}$ which extracts all the low bits in 2-bit regions within x; we also shift x right by one and again mask to extract all the corresponding high bits. These values are added to count the number of bits within the 2-bit regions. We divide-and-conquer again by considering 4-bit regions and so on until we end up with a total for the full 16-bit region.

d. Perhaps the easiest example to understand is the population count function. Since the input x is only 16-bits, there are only 2^{16} potential values the caller can use for x. Thus, we can easily compute $H(x)$ for every one of these inputs, store them

in a table and then look them up rather than running the original function. For example, the C functions could be

```
uint16* T = NULL;

void precomp()
{
  T = ( uint16* )( malloc( 65536 * sizeof( uint16 ) ) );

  for( int i = 0; i < 65536; i++ ) {
    uint16 x = i;

    x = ( x & 0x5555 ) + ( ( x >> 1 ) & 0x5555 );
    x = ( x & 0x3333 ) + ( ( x >> 2 ) & 0x3333 );
    x = ( x & 0x0F0F ) + ( ( x >> 4 ) & 0x0F0F );
    x = ( x & 0x00FF ) + ( ( x >> 8 ) & 0x00FF );

    T[ i ] = x;
  }
}

uint16 H( uint16 x )
{
  return T[ x ];
}
```

In the first function, we run through all possible values of x and store the results $H(x)$ in the table T. Then, in the actual function for $H(x)$ we look the value up from the table. Clearly this only produces an improvement if we use the function very often otherwise we have wasted time computing all the possible results; additionally we have made a trade-off in terms of time and space. That is, our pre-computation requires some memory to hold T so although it takes less time per-call to $H(x)$, it requires more space than the initial version.

52 Obviously combinations of the optimisation techniques can be applied, here we simply treat each one separately.

Off-line pre-computation means doing some one-off computation before the function is called so that we can accelerate execution when it is called; since the pre-computation is off-line in this case, it should be possible at compile-time i.e., before the program is even execute. In this context, we can pre-compute a table, say T, where each T_i is the result of applying the function to i. This is possible since the input x is only 16-bits in size: there are only $2^{16} = 65536$ 16-bit entries in the table so although this consumes some memory, it is not a lot of memory. Then, when the function is called we simply look-up T_i rather than doing any actual computation:

```
uint16* T = NULL;

void precomp()
{
  T = ( uint16* )( malloc( 65536 * sizeof( uint16 ) ) );
```

```
  for( int i = 0; i < 65536; i++ )
  {
    uint8 a = ( i      ) & 0xFF;
    uint8 b = ( i >> 8 ) & 0xFF;

    a = a >> 1;
    b = b >> 1;

    T[ i ] = ( uint16 )( a      ) |
             ( uint16 )( b << 8 ) ;
  }
}

void packed_div2_1( uint16* x )
{
  for( int i = 0; i < 3; i++ )
  {
    x[ i ] = T[ x[ i ] ];
  }
}
```

Note that rather than running the pre-computation function at run-time (i.e., when the program is loaded), we could create the table at compile-time and simply load it as a constant array or even embed it in the program source code.

Roughly speaking, specialisation is the act of replacing general purpose operations with special purpose operations that are less expensive. In this function, one example that occurs is division by the constant two; using a general purpose division is expensive (division takes longer than most operations on a typical processor). However, division by 2^k is equivalent to a right-shift by k bits which is less expensive. We can therefore re-write the function:

```
void packed_div2_2( uint16* x )
{
  for( int i = 0; i < 3; i++ )
  {
    uint8 a = ( x[ i ]      ) & 0xFF;
    uint8 b = ( x[ i ] >> 8 ) & 0xFF;

    a = a >> 1;
    b = b >> 1;

    x[ i ] = ( uint16 )( a      ) |
             ( uint16 )( b << 8 ) ;
  }
}
```

Note that in this case, we could also include the shift of b with the shift required to mask the value in the first place; this saves another operation. Also, it is possible to avoid the unpacking and repacking all together by performing the shift in one go and using a mask to patch up the result:

```
void packed_div2_3( uint16* x )
{
  for( int i = 0; i < 3; i++ )
  {
    x[ i ] = ( x[ i ] >> 1 ) & 0x7F7F;
  }
}
```

Optimisation using parallelism is slightly tricky to write down in C, but roughly speaking you have to imagine that the host processor is equipped with the facility for SIMD parallelism: it can execute the same instruction on many data items at once (where the data items are packed into one blob). So if one can perform two divisions at once for example, one can clearly perform the divisions of a and b at the same time and get a speed-up of roughly 2 for this fragment of the function. This could be thought of as similar to this function:

```
void packed_div2_4( uint16* x )
{
  for( int i = 0; i < 3; i++ )
  {
    uint8 a = ( x[ i ]      ) & 0xFF;
    uint8 b = ( x[ i ] >> 8 ) & 0xFF;

    par
    {
      a = a / 2;
      b = b / 2;
    }

    x[ i ] = ( uint16 )( a      ) |
             ( uint16 )( b << 8 ) ;
  }
}
```

where the `par` keyword is denoting parallelism and the accesses to elements in `t` are meant to show that the input to the SIMD operation is naturally packed. For more able processors (e.g., MIMD rather than SIMD), one might actually be able to run the three (independent) iterations of the loop in parallel with each other.

Finally, program restructuring roughly means re-writing the program control flow so that some advantage is gained: optimisations of this type include loop unrolling, loop fusion, loop fission and so on. Loop unrolling is a good candidate in this case since we are paying a high price for executing loop control (such as the test, the counter increment and so on) even though we know there will only ever be three iterations. Therefore, we can eliminate this overhead and re-write the function as:

```
void packed_div2_5( uint16* x )
{
  uint8 a;
  uint8 b;
```

```
  a = ( x[ 0 ]      ) & 0xFF;
  b = ( x[ 0 ] >> 8 ) & 0xFF;

  a = a / 2;
  b = b / 2;

  x[ 0 ] = ( uint16 )( a      ) |
           ( uint16 )( b << 8 ) ;

  a = ( x[ 1 ]      ) & 0xFF;
  b = ( x[ 1 ] >> 8 ) & 0xFF;

  a = a / 2;
  b = b / 2;

  x[ 1 ] = ( uint16 )( a      ) |
           ( uint16 )( b << 8 ) ;

  a = ( x[ 2 ]      ) & 0xFF;
  b = ( x[ 2 ] >> 8 ) & 0xFF;

  a = a / 2;
  b = b / 2;

  x[ 2 ] = ( uint16 )( a      ) |
           ( uint16 )( b << 8 ) ;
}
```

which looks longer but probably executes faster.

53

a. In the fragment, i is the induction variable (i.e., the loop counter) and n is the loop bound (i.e., the number of times the loop body is executed). If we can determine n statically, we can replace the loop with n copies of the loop body where in the j-th copy we replace the induction variable i with the constant j. This is termed "full" loop unrolling; for example with $n = 4$ we would unroll the loop to get

```
A[ 0 ] = B[ 0 ] + C[ 0 ];
A[ 1 ] = B[ 1 ] + C[ 1 ];
A[ 2 ] = B[ 2 ] + C[ 2 ];
A[ 3 ] = B[ 3 ] + C[ 3 ];
```

The unrolled alternative might increase performance because:

i. The overhead of operating the loop (i.e., initialisation, test and update of the induction variable) is eliminated; essentially the unrolled loop requires less instructions to be executed.
ii. At least one branch per-iteration is required to direct control-flow from the bottom of the loop body back to the top again. This is overhead as above, but potentially also causes a problem in pipelined processors if we cannot accurately predict the branch direction and successfully use speculative execution.

By eliminating the branch, this problem is also eliminated and the straight-line unrolled loop might execute faster as a result.

However, it might decrease performance because:

i. Depending on n, the number of instructions required to implement the loop will typically be smaller than the number required to implement the unrolled alternative (although the loop might still execute more instructions at run-time). As a result instructions in the unrolled loop are not as spatially local as in the loop: a significant distance might exist between the first and last instructions for example. This means an instruction cache might be less effective. Furthermore, it might be hard to keep the unrolled loop in the cache since there is more chance of interference. Both features mean there is a chance that performance could be decreased.

b. Continuing from the solution above, if we do not know n or it is too large to permit full unrolling we might "partially" unroll the loop; this means that we reduce the number of iterations but do not totally eliminate the loop. For example, imagine $n = 400$: we probably wouldn't want to fully unroll the loop in this case but could instead partly unroll by a factor of say $m = 4$ to get:

```
for( int i = 0; i < 400; i += 4 )
{
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
  A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}
```

Now consider the case of $n = 402$. If we use the same method as above, we end up with two extra iterations that we cannot accommodate in the main loop since m does not exactly divide n. We need to add a loop epilogue after the main loop to cope with the cases of $i = 400$ and $i = 401$:

```
for( int i = 0; i < 400; i += 4 ) {
  A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
  A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
  A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
  A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
}

A[ 400 ] = B[ 400 ] + C[ 400 ];
A[ 401 ] = B[ 401 ] + C[ 401 ];
```

or alternatively we could add a loop prologue before the main loop:

```
A[ 0 ] = B[ 0 ] + C[ 0 ];
A[ 1 ] = B[ 1 ] + C[ 1 ];

for( int i = 2; i < 402; i += 4 ) {
```

```
    A[ i+0 ] = B[ i+0 ] + C[ i+0 ];
    A[ i+1 ] = B[ i+1 ] + C[ i+1 ];
    A[ i+2 ] = B[ i+2 ] + C[ i+2 ];
    A[ i+3 ] = B[ i+3 ] + C[ i+3 ];
  }
```

References

1. A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006. ISBN: 0-321-48681-1.
2. A.W. Appel and M. Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 2004. ISBN: 0-521-60765-5.
3. T. Ball and J.R. Larus. Branch Prediction For Free. In *Programming Language Design and Implementation (PLDI)*, 300–313, 1993.
4. G. Boole. *An Investigation of the Laws of Thought*, 1854.
5. A.D. Booth. A Signed Binary Multiplication Technique. In *Quarterly Journal of Mechanics and Applied Mathematics*, **4** (2), 236–240, 1951.
6. P. Briggs, K. Cooper and L. Torczon. Improvements to Graph Coloring Register Allocation. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **16** (3), 428–455, 1994.
7. R.A. Brooker. The Programming Strategy used with the Manchester University Mark 1 Computer. Proceedings of the IEE, **103** (B), 151–157, 1956.
8. R.A. Brooker. An Attempt to Simplify Coding for the Manchester Electronic Computer. British Journal of Applied Physics **6**, 307–311, 1955.
9. A.W. Burks A.R. Burks. The ENIAC: The First General-Purpose Electronic Computer. In *IEEE Annals of the History of Computing*, **3** (4), 310–389, 1981.
10. B. Calder, C. Krintz, S. John and T. Austin. Cache-Conscious Data Placement. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 139–149, 1998.
11. M. Campbell-Kelly. Programming the EDSAC: Early programming activity at the University of Cambridge. In *IEEE Annals of the History of Computing*, **2** (1), 7–36, 1980.
12. P.E. Ceruzzi. A History of Modern Computing. MIT Press, 2003. ISBN: 0-262-53203-4.
13. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins and P.W. Markstein. Register Allocation via Colouring. In *Computer Languages*, **6**, 47–57, 1981.
14. W.W. Chandler. The Installation and Maintenance of Colossus In *IEEE Annals of the History of Computing*, **5** (3), 260–262, 1983.
15. B. Chapman, G. Jost and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007. ISBN: 0-262-53302-2.
16. T.M. Chilimbi, M.D. Hill, J.R. Larus. Cache-Conscious Structure Layout. In *Programming Language Design and Implementation (PLDI)*, 1–12, 1999.
17. J. Cocke and V. Markstein. The Evolution of RISC Technology at IBM. IBM Journal of Research and Development, **34** (1), 4–11, 1990.
18. A.W.M. Coombs. The Making of Colossus. In *IEEE Annals of the History of Computing*, **5** (3), 253–259, 1983.
19. L. Dadda. Some Schemes for Parallel Multipliers. In *Alta Frequenza*, **34**, 349–356, 1965.
20. R. Espasa, M. Valero, J.E. Smith. Vector Architectures: Past, Present and Future. In *International Conference on Supercomputing*, 425–432, 1998.

21. J.A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *International Symposium on Computer Architecture (ISCA)*, 140–150, 1983.
22. K. Flautner, N.S. Kim, S. Martin, D. Blaauw and T.N. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *International Symposium on Computer Architecture (ISCA)*, 148–157, 2002.
23. T.H. Flowers. The Design of Colossus. In *IEEE Annals of the History of Computing*, **5** (3), 239–252, 1983.
24. M.J. Flynn. Some Computer Organizations and Their Effectiveness. In *IEEE Transactions on Computers*, **21** (9), 948–960, 1972.
25. J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. In *Communications of the ACM*, **4** (10), 435–436, 1961.
26. A.H. Hashemi and D.R. Kaeli and B. Calder. Efficient Procedure Mapping Using Cache Line Coloring. In *Programming Language Design and Implementation (PLDI)*, 171–182, 1997.
27. J.L. Hennessy. RISC-Based Processors: Concepts and Prospects. In *New Frontiers in Computer Architecture*, 95–103, 1986.
28. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2006. ISBN: 0-123-70490-1.
29. M.D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987.
30. N.P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *International Symposium on Computer Architecture (ISCA)*, 364–373, 1990.
31. N.S. Kim, K. Flautner, D. Blaauw and T.N. Mudge. Drowsy Instruction Caches: Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction. In *International Symposium on Microarchitecture (MICRO)*, 219–230, 2002.
32. D.E. Knuth. Fundamental Algorithms. The Art of Computer Programming **1** 3rd Edition, Addison-Wesley, 1998.
33. D.E. Knuth. Seminumerical Algorithms. The Art of Computer Programming **2** 3rd Edition, Addison-Wesley, 1998.
34. D.E. Knuth. Sorting and Searching. The Art of Computer Programming **3** 2nd Edition, Addison-Wesley, 1998.
35. J.R. Larus. SPIM: A MIPS32 Simulator. Available at: `http://pages.cs.wisc.edu/~larus/spim.html`
36. S.H. Lavington. Manchester Computer Architectures, 1948-1975. In *IEEE Annals of the History of Computing*, **15** (3), 44–54, 1993.
37. J.R. Levine. *Linkers and Loaders*. Morgan-Kaufmann, 2000. ISBN: 1-558-60496-0.
38. M.S. Malone. *Infinite Loop: How Apple, the World's Most Insanely Great Computer Company, Went Insane*. Aurum Press, 1999. ISBN: 1-854-10693-7.
39. E.J. McCluskey. Minimization of Boolean Functions. In *Bell System Technical Journal*, **35** (5), 1417–1444, 1956.
40. S. McFarling and J. Hennesey. Reducing the Cost of Branches. In *International Symposium on Computer Architecture (ISCA)*, 396–403, 1986.
41. F.H. McMahon. The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, 1986.
42. MIPS Technologies. *MIPS32 Architecture For Programmers Volume I : Introduction to the MIPS32 Architecture*. Technical Report MD00082, 2001.
43. MIPS Technologies. *MIPS32 Architecture For Programmers Volume II : The MIPS32 Instruction Set*. Technical Report MD00086, 2001.
44. MIPS Technologies. *MIPS32 Architecture For Programmers Volume III: The MIPS32 Privileged Resource Architecture*. Technical Report MD00090, 2001.
45. G. Moody. *Rebel Code: Linux and the Open Source Revolution*. Penguin Press, 2001. ISBN: 0-713-99520-3.
46. G.E. Moore. Cramming more components onto integrated circuits. In *Electronics Magazine*, **38** (8), 114–117, 1965.

47. S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, 1997. ISBN: 1-558-60320-4.
48. J. von Neumann. First Draft of a Report on the EDVAC. In *IEEE Annals of the History of Computing*, **15** (4), 27–75, 1993.
49. P.S. Pacheco. Parallel Programming with MPI. Morgan-Kaufmann, 1997. ISBN: 1-558-60339-5.
50. D.A. Patterson. Reduced Instruction Set Computers. In *Communications of the ACM*, **28** (1), 8–21, 1985.
51. K. Pettis and R.C. Hansen. Profile Guided Code Positioning. In *Programming Language Design and Implementation (PLDI)*, 16–27, 1990.
52. S. Peyton Jones, N. Ramsey and F. Reig. C—: A Portable Assembly Language that Supports Garbage Collection. In *Principles and Practice of Declarative Programming (PPDP)*, 1–28, 1999.
53. M. Poletto and V. Sarkar. Linear Scan Register Allocation. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **21** (5), 895–913, 1999.
54. M. Powell and S.-H. Yang and B. Falsafi and K. Roy and T. N. Vijaykumar. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *ACM/IEEE International Symposium on Low Power Electronics and Design*, 90–95, 2000.
55. W.V. Quine. A Way to Simplify Truth Functions, In *American Mathematical Monthly*, **62** (9), 627–631, 1995.
56. B. Randell. The Origins of Digital Computers: Selected Papers. Springer-Verlag, 1982. ISBN: 0-387-11319-3.
57. J. Raskin. *The Humane Interface: New Directions for Designing Interactive*. Addison-Wesley, 2000. ISBN: 0-201-37937-6.
58. E. Rotenberg, S. Bennett and J.E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *International Symposium on Microarchitecture (MICRO)*, 24–35, 1996.
59. J. Rymarczyk. Coding Guidelines for Pipelined Processors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*, 12–19, 1982.
60. C.E. Shannon. The Symbolic Analysis of Relay and Switching Circuits. MSc Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1940.
61. J.E. Smith. A Study of Branch Prediction Strategies. In *International Symposium on Computer Architecture (ISCA)*, 135–148, 1981.
62. R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. In *IBM Journal of Research and Development*, **11** (1), 25–33, 1967.
63. C.S. Wallace. A Suggestion for Fast Multipliers. In *IEEE Transactions on Computers*, **13** (1), 14–17, 1964.
64. H.S. Warren Jr. *Hackers Delight*. Addison-Wesley, 2003. ISBN: 0-201-91465-4.
65. D.J. Wheeler. Programme Organization and Initial Orders for the EDSAC. In *Proceedings of the Royal Society of London: Series A, Mathematical and Physical Sciences*, **202** (1071), 573–589, 1950.
66. M.V. Wilkes. Arithmetic on the EDSAC. In *IEEE Annals of the History of Computing*, **19** (1), 13–15, 1997.
67. M.V. Wilkes. Memoirs of a Computer Pioneer. MIT Press, 1985. ISBN: 0-262-23122-0.
68. W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implications of the Obvious. In *ACM SIGARCH Computer Architecture News*, **23** (1), 20–24, 1995.
69. T-Y. Yeh and Y.N. Patt. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. In *International Symposium on Computer Architecture (ISCA)*, 257–266, 1993.

Index

3-state logic, 51
3 : 2 compressor, 254
GND, 46
V_{dd}, 46
n-to-*m*, 69
n-tuple, 17

A-0, 399
accumulator, 144
accumulator machine, 184
accumulators, 144
active high, 79
active low, 79
active process, 523
address, 148, 179, 270
address bus, 177
address counter, 418
address decoder, 286
address space, 511
Address Space Number (ASN), 512
address translation, 302
addressing, 179
addressing mode, 179
addressing modes, 180
advances, 340
aliases, 418, 429
aligned, 281
Amdahl's Law, 219
American Standard Code for Information Interchange (ASCII), 32
anti-dependence, 337
architectural register, 371
Arithmetic and Logic Unit (ALU), 144, 223
arithmetic shifts, 237
assembler, 417
assemblers, 397, 398
assembly language, 417
assigned, 3
assigning, 4
associativity, 310
atomic number, 43
atoms, 43
Autocode, 399

back-end, 360, 452
backed, 371, 373
backward reference, 439
bag, 10
bandwidth, 214, 341
Banker's Rounding, 37
banking, 289
base, 23
basic block, 455
Basic Input/Output System (BIOS), 501
benchmark, 218
biasing, 36
big-endian, 28
bijective, 15
binary, 24
Binary Coded Decimal (BCD), 30
binary operation, 19
binary point, 24
binary relation, 17
binary vector, 27
binary vectors, 27
Bipolar Junction Transistors (BJT), 46
bit, 27
bit-serial shifter, 238
bit-slicing, 67
bit-vector, 27
blocking assignments, 119
body, 101
Bombe, 145
bonding wires, 89

boot-loader, 502
boot-strap, 501
Booth encoding, 257
booting, 502
bootstrapped, 454
bootstrapping, 453
bound, 7
branch, 192
branch delay slot, 356
branch prediction, 358
Branch Prediction Buffer (BPB), 359
branch table, 479
Branch Target Buffer (BTB), 359
branch target prediction, 358
Branch Unit (BU), 369
breakpoint, 565
bubble, 352
buffer, 61
bundle, 389
burst mode, 283
bus, 154, 177
bus cycles, 178
bus protocol, 154, 178
bus width, 177
byte, 29
byte addressed, 279

cache, 300
cache bypass, 315
cache coherency, 334
cache contention, 310
cache interference, 310
cache lines, 301
cache pollution, 304
cache-hits, 301
cache-misses, 301
callee, 479
caller, 479
calling convention, 482
Capacity misses, 301
cardinality, 10
carry look-ahead adder, 232
carry-save, 254
Cartesian product, 17
cascading, 68
ceiling, 38
cell, 46
Central Processing Unit (CPU), 143
channel, 389
characters, 32
charge, 44
child process, 526
chip, 89
Clamp, 230
clamped addition, 388
clock, 75
clock cycle, 75
clock edges, 75
clock frequency, 75
clock period, 75
clock pulse, 75
clock rate, 75
clock signal, 75
clock skew, 76
clocked SR-latch, 79
code bloat, 552
code density, 189
code generation, 452
code word, 298
codomain, 14, 15
Colossus, 144
colouring stack, 460
column-major, 548
combinatorial logic, 48
comment, 104
Common Data Bus (CDB), 371
communication model, 333
Compact Disc Read Only Memory (CDROM), 297
Compact Disc Read-Write (CDRW), 297
compilation tool, 397
compiler tool-chain, 397
compilers, 397, 451
complement, 13
Complementary Metal-Oxide Semiconductor (CMOS), 46
Complex Instruction Set Computers (CISC), 199
composition, 16
Compulsory misses, 301
computational model, 332
concatenation, 28, 112
conclusion, 6
concurrency, 331
conductors, 44
Conflict misses, 301
conjunctive normal form, 22
connective, 5
consistency, 304
Constant Angular Velocity (CAV), 297
Constant Linear Velocity (CLV), 297
context, 522
context switch, 523
continuous, 107
continuous assignment, 110
contradiction, 7
control-flow, 161, 192
Control-Flow Graph (CFG), 457

control-path, 154
cooperative multitasking, 527
cost function, 213
critical path, 61
cross compiler, 454
Cycles Per-Instruction (CPI), 215
Cyclic Redundancy Check (CRC), 298
cylinder, 295

D-type latch, 79
Dadda multiplier, 255
data bus, 177
data order, 335
data section, 400
Data-Flow Graph (DFG), 471
data-path, 154
Debug Mode , 500
decimal, 24
decimal point, 24
decoder, 69
define-use dependency, 337
definition set, 458
delay, 114
demand paging, 517
demultiplexer, 67
denary, 24
denominator, 14
dependent, 336
design automation, 98
device driver, 506
die, 89
difference, 13
digit set, 23
Digital Versatile Disc (DVD), 297
direct addressing, 181
direct-mapped cache, 305
directive, 431
directives, 418
dirty, 305
disjoint, 13
disjunctive normal form, 22
disk head, 295
disk map, 517
dispatched, 371
domain, 14
don't care state, 57
donor, 44
doping, 44
double-word, 29
drain, 45
drive, 102
drowsy cache, 320
duty cycle, 75
dynamic prediction, 358
Dynamic Random Access Memory (DRAM), 292
dynamic scheduling, 335, 360

early exit, 251
early termination, 251
edge triggered, 80
effective address, 181
electron configuration, 43
Electronic Delay Storage Automatic Calculator (EDSAC), 158
Electronic Design Automation (EDA), 98
Electronic Discrete Variable Automatic Computer (EDVAC), 150
Electronic Numerical Integrator And Computer (ENIAC), 146
electrons, 43
element, 270
elements, 10
empty set, 10
emulator, 561
enable, 70
enable gate, 51
encoder, 69
endianness, 28, 280
Enigma machine, 145
equality, 66
equivalence relation, 17
equivalent, 7
error, 427
error correction, 297
error detection, 297
essential prime implicants, 59
etching, 88
evaluated, 4
even parity, 298
event management, 496
evict, 303
exception, 427, 502
exception vector, 505
executable file, 407
execution table, 366
execution time, 214
existential quantifier, 7
expanding opcode, 191
expansion, 23
explicit branch, 457
exponent, 34, 35
expression, 5

Fan-in, 63
fan-out, 63
fetch-decode-execute cycle, 155
fetched, 301, 303

Field Effect Transistor (FET), 44
Field Programmable Gate Array (FPGA), 92
Finite State Machine (FSM), 81
First-In First-Out (FIFO), 368
First-In Last-Out (FILO), 401
fixed point, 34
fixed prediction, 358
flip-flop, 80
floating point, 35
Floating Point Unit (FPU), 224, 383
floor, 38
flush, 358
Flynn's Taxonomy, 332
forward reference, 439
forward referencing problem, 439
fragmentation, 403
frame pointer, 488
frame table, 517
frames, 297, 515
free store, 403
free variable, 7
front-end, 360, 451
full-adder, 65
fully vectored, 505
fully-associative cache, 310
function, 14
function behaviour, 15
function epilogue, 482
function inlining, 551
function prologue, 482
function signature, 15
functions, 126

gate, 45
gate delay, 61
gated-V_{dd} cache, 320
generate statement, 133
graph colouring, 458
Graphical User Interface (GUI), 496
Gray code, 55
Greatest Common Divisor (GCD), 195
guarded, 194

half-adder, 63
half-word, 29
Hamming Code, 299
Hamming distance, 24
Hamming weight, 24
handler, 502
handshake, 282
hard errors, 297
Hardware Description Language (HDL), 98
Harvard architecture, 148
hazard, 338
heap, 403
hexadecimal, 24
hidden digit, 35
hierarchy, 108
high impedance, 51, 103
hit ratio, 302
hoisting, 546
holes, 44
hyperthreading, 379
hypothesis, 6

identifier, 101
identity elements, 19
identity function, 16
idle process, 526
image, 14
immutable, 270
implicant, 57
implicit branch, 457
imprecise, 503
in-flight, 348
in-order, 336
in-order completion, 373
in-order dispatch, 372
in-order issue, 368
index, 148, 270
indexed addressing, 182
induction variable, 543
Initial Orders, 158
injective, 15
instantiated, 105
instruction classes, 185
instruction format, 186
instruction mix, 215
instruction pointer, 154
Instruction Register (*IR*), 172
instruction scheduling, 452, 468
instruction selection, 452, 468
instruction set, 149
instruction set architecture, 171
Instruction Set Architecture (ISA), 154
instruction table, 366
Instruction-Level Parallelism (ILP), 331
instructions, 144
insulators, 44
integers, 14
inter-sector gap, 295
interface, 100
interference graph, 460
interleaved access, 289
intermediate instructions, 455
intermediate program, 455
interrupt, 502
interrupt mask, 503

interrupt request, 502, 504
intersection, 13
intra-assignment delay, 120
inverse, 16
inversion bubble, 50
ion, 44
ionisation, 44
irrational, 14
issue, 349
issue buffer, 361, 367

JK-latch, 79
jump, 192

Karnaugh map, 54
kernel, 496, 536
kernel functions, 506
kernel memory, 500
Kernel Mode , 500

lands, 296
lane, 389
lanes, 383
latency, 214, 341
leakage, 45
least-significant, 28
less than, 66
level, 75
level triggered, 80
library, 398
linkers, 397, 398
lithography, 87
little-endian, 28
live range, 464
live-in, 458
live-out, 458
load modules, 407
load point, 164
load-use dependency, 337
Load/Store Unit (LSU), 368
loader, 398
locality, 290
location, 270
logarithmic shifter, 241
logic gates, 47
logic styles, 45
logical shifts, 237
long-term, 270
Look-Up Table (LUT), 92
loop blocking, 549, 550
loop bound, 545
loop epilogue, 546
loop fission, 544
loop fusion, 543
loop interchange, 549
loop invariant, 546
loop prologue, 546
loop unrolling, 545
loops, 192
Lorenz machine, 144
low-level format, 295

machine code, 397
machine instructions, 418
mantissa, 34, 35
mask, 87
maskable, 503
mastering, 297
maxterms, 22
mean access time, 302
member, 10
memory, 144, 269
Memory Address Register (*MAR*), 172
memory bank, 289
Memory Buffer Register (*MBR*), 172
memory cells, 283
memory consistency, 336
memory hierarchy, 277
memory latency, 282
memory management, 496
Memory Management Unit (MMU), 514
memory model, 400
memory transaction, 282
memory wall, 160, 270
mercury delay line, 271
message passing architecture, 333
meta-data, 408
Metal Oxide Semiconductor Field-Effect Transistor (MOSFET), 45
metastability, 61
metrics, 213
micro-architecture, 171
micro-coded, 177
Micro-Processor without Interlocked Pipeline Stages (MIPS), 169
microchip, 89
Millions of Instructions Per Second (MIPS), 214
minterms, 22
mispredict, 358
misprediction rate, 358
miss ratio, 302
models, 99
modified Booth recoding, 258
module, 100
most-significant, 28
multi-set, 10

Multiple Instruction, Multiple Data (MIMD), 333
Multiple Instruction, Single Data (MISD), 333
multiple-issue, 360, 389
multiplexer, 67
multiplicand, 243
multiplier, 243
multiprocessor, 334
multiprogramming, 527
multitasking, 522
mutable, 270
mutually exclusive, 13

Named port lists, 132
native compiler, 454
natural numbers, 14
nested-`if`, 475
net, 101
netlist, 99
neutrons, 43
nibble, 29
non-active processes, 523
non-blocking assignments, 119
non-maskable, 503
Non-Uniform Memory Access (NUMA), 334
non-volatile, 270
normalised, 35
not-a-number, 37
nucleus, 43
null set, 10
numerator, 14

object file, 407
object modules, 407
octal, 24
odd parity, 298
off-chip caches, 315
on-chip caches, 315
one-hot coding, 86
one-of-many, 69
opcode, 149
opcodes, 144
operands, 144, 149
Operating System (OS), 495
orphan process, 527
out-of-order, 336
out-of-order dispatch, 372
output dependency, 338

package, 89
pads, 89
page fault, 520
page table, 514, 516
paged virtual memory, 513
pages, 514, 515
pair, 17
paper tape, 271
parallel block, 117
parallelism, 331
parametrisation, 131
parent process, 526
parity code, 298
partial unrolling, 545
partly vectored, 505
pass-by-register, 486
pass-by-stack, 486
peephole optimisation, 434
performance, 213, 214
peripherals, 144
permanent, 297
photo-lithography, 88
photo-resist, 88
physical addresses, 512
physical memory, 512
physical register, 371
pins, 89
pipeline cycle, 340
pipeline register, 341
pipeline stages, 340
pipelined processors, 331
pipelining, 339
pits, 296
plane, 90
platters, 294
polling loop, 502
port, 101
port list, 101
Position Independent Code (PIC), 416
pre-processor, 129
precise, 503
predecessor set, 458
predicate, 3
predicate function, 8
predicated, 194
preemptive multitasking, 527
preimage, 14
primary, 270
prime implicants, 59
primitive modules, 105
principle of inclusion, 13
priority encoder, 71
priority level, 523
privilege levels, 499
Privileged Resource Architecture (PRA), 497
procedural assignment, 119
process, 117, 400, 522
Process Control Block (PCB), 523
Process Identifier (PID), 523

process management, 496
process queue, 525
process scheduling, 524
process states, 524
Processing Elements (PEs), 331
processor modes, 499
Product of Sums (PoS), 22
program consistency, 336
Program Counter (*PC*), 154, 172
program image, 407
program order, 335
Programmable Logic Array (PLA), 90
propagation delay, 61
proper sub-set, 13
proper super-set, 13
proposition, 3
protons, 43
pseudo-instructions, 418, 429
Pseudo-Static RAM, 292
pull down network, 46
pull up network, 46
pure vector, 380

quality, 213
quantifier, 7
Quine-McCluskey minimisation, 57

radix, 23
Rainbow Book, 297
raised, 504
RAMs, 270
random-access, 270
range, 15
rational numbers, 14
re-entrant, 505
Re-Order Buffer (ROB), 361, 371, 372
read enable, 287
read head, 296
read port, 286
Read-Only (RO), 270
Read-Write (RW), 270
recoding, 257
Red Book, 297
Reduced Instruction Set Computers (RISC), 199
reduction, 114
reflexive, 17
register allocation, 424, 536
register file, 172
register machine, 184
register pressure, 464
register renaming, 371, 373
register table, 366
registers, 115, 144, 172
regular delay, 114
relocate, 164
relocated, 416
relocation, 400
relocation entries, 409
rename table, 373
replication, 112
reservation stations, 361, 370
resident, 303, 522
resolution, 279
resolved, 192, 417
retired, 350
return address, 197
ripple-carry adder, 231
ROMs, 270
rotational latency, 296
rotations, 237
rounding, 37
rounding mode, 37
row-major, 548

saturating addition, 388
saturating counter, 359
scheduler, 524
scheduling, 335
scoreboard, 361
secondary, 270
sectors, 295, 297
seek time, 295
selection, 112
self-hosted, 453
semantics, 154
semiconductors, 44
sensitivity list, 118
sequenced-`if`, 475
sequential block, 117
sequential-access, 270
set, 10
set-associative cache, 312
sets, 312
shells, 43
shelve, 371
shift, 236
short-circuiting, 540
short-term, 270
sign bit, 35
sign extension, 188
sign-magnitude, 30
simulator, 561
Simultaneous Multi-Threading (SMT), 379
Single Instruction, Multiple Data (SIMD), 332
Single Instruction, Single Data (SISD), 332
Single Program, Multiple Data (SPMD), 333
single stepping, 564

single vectored, 505
single-issue, 349, 382
Small-Scale Experimental Machine (SSEM), 158
soft errors, 297
software pipelining, 554
source, 45
Spatial locality, 290
special instructions, 429
speculative execution, 357
spill, 464, 480
spill heuristic, 464
SPIM, 561
squashed, 358
SR-latch, 77
stack, 401
stack frame, 481
stack machine, 184
Stack Pointer, 401
stack relative addressing, 480
stack section, 400
stall, 352
static branch prediction, 470
static prediction, 358
Static Random Access Memory (SRAM), 291
static scheduling, 335, 382, 389
storage device, 269
storage hierarchy, 277
stored program architecture, 153
straight-line code, 541
stride, 383
string, 32
strongly consistent, 336
sub-set, 12
sub-word, 382
sub-words, 301
substrate, 87
successor set, 458
Sum of Products (SoP), 22
super-set, 12
superscalar, 360
superscalar processors, 331
Supervisor Mode, 500
surjective, 15
swap file, 514
switched, 46
switches, 44
symbol table, 440
symbols, 410
symmetric, 17
system calls, 506
system task, 128

table driven assembler, 441
tag, 301
tail-end recursive, 553
tasks, 126
tautology, 7
Temporal locality, 290
terms, 5
text section, 400
thrashing, 520
thread, 522
throughput, 214, 341
tiling, 468
time quantum, 527
time sharing, 527
time slice, 527
time slicing, 527
Time Stamp Counter (TSC), 218
timer interrupt, 503
timing, 114
TLB-hit, 522
TLB-miss, 522
toggled, 46
tool-chain, 99
Top of Stack (*ToS*), 401
trace, 378, 457
trace cache, 379
trace scheduling, 470
tracks, 295
transient, 297
Transistor-Transistor Logic (TTL), 46
transistors, 44
transitive, 17
Translation Look-aside Buffer (TLB), 521
trap, 502
tree multiplier, 249
trigger, 118
Truncate, 230
truth table, 5
truth value, 3
tuple, 17
Turing-complete, 146
two-level predictor, 359
twos-complement, 31

unaligned, 281
unary operation, 19
undefined, 103
unicode, 34
Uniform Memory Access (UMA), 334
union, 13
unit of transfer, 283
universal, 50
universal quantifier, 7
universal set, 11
unknown, 103

usage set, 458
user functions, 506
user memory, 500
User Mode , 500
User-Defined Primitive (UDP), 108

vacuum tube, 271
variable length encoding, 189
variables, 3
Vector Processing Unit (VPU), 383
vector processors, 331, 380
vector register, 382
vector register file, 382
vectorial, 22
vectorisation, 386
vectors, 17
Venn diagram, 13
Verilog, 98
Very Long Instruction Word (VLIW), 389
victim cache, 316
Video CDs (VCDs), 297
virtual addresses, 512
virtual memory, 512
volatile, 270
von Neumann architecture, 153
von Neumann bottleneck, 160, 270

wall-clock time, 216
Wallace multiplier, 254
weakly consistent, 336
White Book, 297
wire, 101
wire delay, 61
wire vectors, 102
word enable, 287
word select, 286
words, 29
working set, 291, 512
workload characterisation, 215
write enable, 287
write port, 286

Yellow Book, 297
yield, 89, 527

zombie process, 526, 527